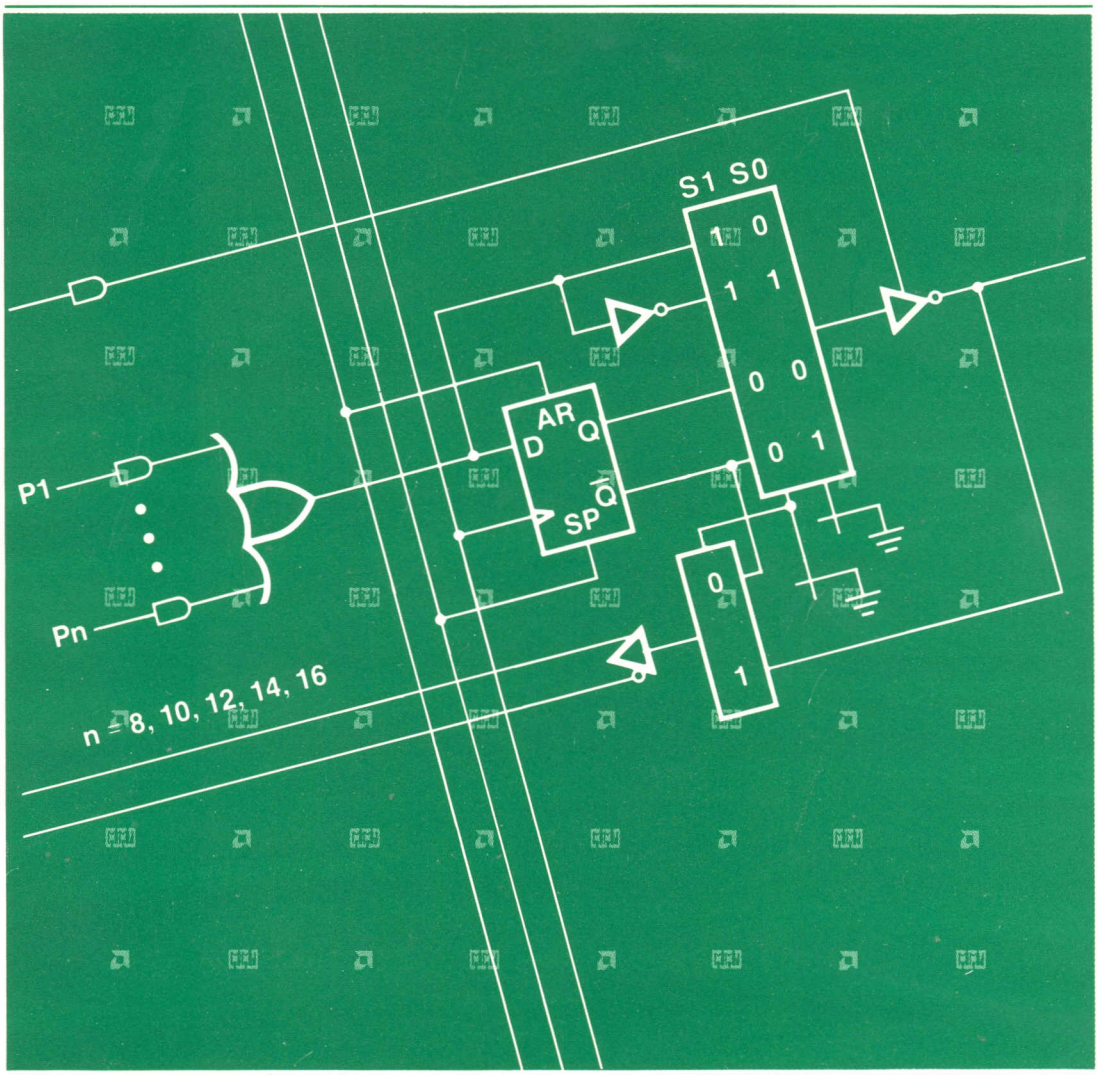
PAL® Device Handbook

Advanced Micro Devices



Monolithic Memories

A Wholly Owned Subsidiary of Advanced Micro Devices

24-pin Combinatorial TTL/CMOS PAL Devices

PRODUCT	PINS	TECHNOLOGY	t_{PD} (ns)	I_{CC} (mA)	DESCRIPTION	PAGE
AmPAL22XP10	24	TTL	20, 30, 40	90, 180, 210	XOR gate	5-271
PAL20S10	24	TTL	35	240	Product term steering	5-103
AmPAL22P10	24	TTL	15, 25	105, 210	24-pin superset	5-291
Am/PAL20L10	24	TTL	15, 20, 25, 30	105, 165, 210	10 outputs	5-113, 5-306
PAL20L8	24	TTL, E CMOS	15, 25, 35, 45	0.1, 105, 210	Standard	5-122
PAL6L16	24	TTL	25	90	Wide output	5-141
PAL8L14	24	TTL	25	90	Wide output	5-141
PAL12L10	24	TTL	40	100	Simple combinatorial	5-147
PAL14L8	24	TTL	40	100	Simple combinatorial	5-147
PAL16L6	24	TTL	40	100	Simple combinatorial	5-147
PAL18L4	24	TTL	40	100	Simple combinatorial	5-147
PAL20L2	24	TTL	40	100	Simple combinatorial	5-147
PAL20C1	24	TTL	40	100	Simple combinatorial	5-147

24-pin Registered TTL/CMOS PAL Devices

PRODUCT	PINS	TECHNOLOGY	f_{MAX} (MHz)	I_{CC} (mA)	DESCRIPTION	PAGE
AmPALC29MA16	24	EE CMOS	20, 15	120	Advanced Async. Macro	5-209
AmPALC29M16	24	EE CMOS	20, 15	120	Advanced Macrocell	5-231
PAL32VX10	24	TTL	25, 22	180	J-K, varied terms	5-70
Am/PAL22V10	24	TTL, E CMOS	40, 33, 28.5, 20, 18	90, 180	Versatile	5-79, 5-249
PAL22RX8	24	TTL	28.5	210	J-K flip-flops	5-87
PAL20RA10	24	TTL	20	200	Asynchronous	5-95
AmPAL20XRP10	24	TTL	30, 22, 14	105, 180, 210	XOR gate & polarity	5-271
AmPAL20XRP8	24	TTL	30, 22, 14	105, 180, 210	XOR gate & polarity	5-271
AmPAL20XRP6	24	TTL	30, 22, 14	105, 180, 210	XOR gate & polarity	5-271
AmPAL20XRP4	24	TTL	30, 22, 14	105, 180, 210	XOR gate & polarity	5-271
PAL20RS10	24	TTL	20	240	Product term steering	5-103
PAL20RS8	24	TTL	20	240	Product term steering	5-103
PAL20RS4	24	TTL	20	240	Product term steering	5-103
PAL20X10	24	TTL	22	180	XOR gate	5-113
PAL20X8	24	TTL	22	180	XOR gate	5-113
PAL20X4	24	TTL	22	180	XOR gate	5-113
AmPAL20RP10	24	TTL	37, 25	105, 210	Programmable polarity	5-291
AmPAL20RP8	24	TTL	37, 25	105, 210	Programmable polarity	5-291
AmPAL20RP6	24	TTL	37, 25	105, 210	Programmable polarity	5-291
AmPAL20RP4	24	TTL	37, 25	105, 210	Programmable polarity	5-291
PAL20R8	24	TTL, E CMOS	37, 25, 20, 15	0.1, 105, 210	Standard	5-122
PAL20R6	24	TTL, E CMOS	37, 25, 20, 15	0.1, 105, 210	Standard	5-122
PAL20R4	24	TTL, E CMOS	37, 25, 20, 15	0.1, 105, 210	Standard	5-122
PAL32R16	40	TTL	16	280	MegaPAL™ device	5-158

(Continued on back cover)

1988

20-pin Combinatorial TTL/CMOS PAL Devices

PRODUCT	PINS	TECHNOLOGY	t_{PD}(ns)	I_{CC}(mA)	DESCRIPTION	PAGE
AmPAL18P8	20	TTL	15, 25	55, 90, 180	20-pin superset	5-202
PAL16P8	20	TTL	25	180	Programmable polarity	5-17
Am/PAL16L8	20	TTL, E CMOS	10, 15, 25, 35, 55	0.1, 45, 90, 180	Standard	5-26, 5-184
PAL10H8	20	TTL	35	90	Simple combinatorial	5-56
PAL10L8	20	TTL	35	90	Simple combinatorial	5-56
PAL12H6	20	TTL	35	90	Simple combinatorial	5-56
PAL12L6	20	TTL	35	90	Simple combinatorial	5-56
PAL14H4	20	TTL	35	90	Simple combinatorial	5-56
PAL14L4	20	TTL	35	90	Simple combinatorial	5-56
PAL16H2	20	TTL	35	90	Simple combinatorial	5-56
PAL16L2	20	TTL	35	90	Simple combinatorial	5-56
PAL16C1	20	TTL	40	0	Simple combinatorial	5-56

20-pin Registered TTL/CMOS PAL Devices

PRODUCT	PINS	TECHNOLOGY	f_{MAX}(MHz)	I_{CC}(mA)	DESCRIPTION	PAGE
AmPAL23S8	20	TTL	33, 28.5	200	Buried registers	5-169
PAL16RA8	20	TTL	20	170	Asynchronous	5-11
PAL16RP8	20	TTL	25	180	Programmable polarity	5-17
PAL16RP6	20	TTL	25	180	Programmable polarity	5-17
PAL16RP4	20	TTL	25	180	Programmable polarity	5-17
Am/PAL16R8	20	TTL, E CMOS	58, 40, 28.5, 18, 11	0.1, 45, 90, 180	Standard	5-26, 5-184
Am/PAL16R6	20	TTL, E CMOS	58, 40, 28.5, 18, 11	0.1, 45, 90, 180	Standard	5-26, 5-184
Am/PAL16R4	20	TTL, E CMOS	58, 40, 28.5, 18, 11	0.1, 45, 90, 180	Standard	5-26, 5-184
PAL16X4	20	TTL	14	225	Arithmetic	5-51

Sequencers

PRODUCT	PINS	TECHNOLOGY	f_{MAX}(MHz)	I_{CC}(mA)	DESCRIPTION	PAGE
PMS14R21	24	TTL	30, 25	210	PROSE Sequencer	5-315
PLS167	24	TTL	33	200	Programmable Logic Seq.	5-331
PLS168	24	TTL	33	200	Programmable Logic Seq.	5-331
PLS105	24	TTL	37	200	Programmable Logic Seq.	5-331
Am29PL141	28	TTL	20	450	Fuse Prog. Controller	5-339
Am2971	24	TTL	85	310	Prog. Event Generator	5-365

ECL PAL Devices

PRODUCT	PINS	TECHNOLOGY	f_{MAX}/t_{PD}	I_{CC}(mA)	DESCRIPTION	PAGE
PAL10H20EV/EG8	24	ECL 10KH	125 MHz	220	Registered/latched	5-381
PAL10020EV/EG8	24	ECL 100K	125 MHz	220	Registered/latched	5-381
PAL10H20G8	24	ECL 10KH	6 ns	225	Latched	5-382
PAL10H20P8	24	ECL 10KH	6 ns	210	Combinatorial	5-386

Logic Cell Array

PRODUCT	PINS	TECHNOLOGY	f_{MAX}(MHz)	I_{CC}(mA)	DESCRIPTION	PAGE
M2064	48,68	RAM CMOS	70, 50, 33	5	Logic Cell Array	5-483
M2018	68,84	RAM CMOS	50, 33	5	Logic Cell Array	5-483

(Continued on front cover)

PAL Device Handbook

PAL® Device Handbook

Introduction	1
Applications	2

PAL Device Data Book

Programming and Quality	3
PALASM® 2 Software User Documentation	4
Data Sheets	5
Appendices	6

© 1988 Advanced Micro Devices, Inc.

Advanced Micro Devices reserves the right to make changes
in its products without notice in order to improve design or performance characteristics.
The performance characteristics listed in this document are guaranteed by specific tests,
correlated testing, guard banding, design and other practices common to the industry.
For specific testing details contact your local AMD sales representative.
The company assumes no responsibility for the use of any circuits described herein.

901 Thompson Place, P.O. Box 3453, Sunnyvale, California 94088
(408) 732-2400 TWX: 910-339-9280 TELEX: 34-6306

In late 1987, the two programmable logic market leaders, Monolithic Memories and Advanced Micro Devices, merged into one great company. We combined our strong traditions of customer service and innovation to offer you the best line of programmable logic devices. The *1988 Data Book* presents the technical specifications on the entire product line. The separate *1988 Handbook* presents all of the necessary support material, whether you are accustomed to using MMI or AMD products.

AMD/MMI has the products, manufacturing capacity and technology to perpetuate the leadership in the programmable logic field which we pioneered. In the *1988 Data Book* you will notice that we not only have the broadest programmable logic line, but we also have the industry's most comprehensive CMOS programmable logic line. We think that you will find the Data Book and Handbook informative and useful. If you have any comments or questions on the books or the product line, please contact us.

Cyrus Tsui

Vice President
Programmable Logic Division

Description

This 1988 PAL Device Handbook/Data Book is your complete guide to all programmable logic devices (PLDs) from Monolithic Memories and Advanced Micro Devices. The merger of the two companies provides a greater wealth of products and services for you. Note that all PLDs which were in production before the merger are still being produced.

The PAL Device Handbook/Data Book is organized into two volumes and six easy-to-use sections:

PAL Device Handbook

Section 1: Introduction
Includes an overview of the PLD product family.

Section 2: Applications
Includes detailed application examples. The first few chapters provide tutorials in PLD design. The application notes are grouped by application area.

PAL Device Data Book

Section 3: Programming and Quality
Includes information on PLD software programs, programming information, PLD technology and quality discussions, and package information.

Section 4: PALASM 2 Software User Documentation
Includes complete documentation for PALASM 2 software.

Section 5: Data Sheets
Includes specifications for all PLDs from the combined company. PAL devices formerly from MMI are under "PAL Devices," while PAL devices formerly from AMD are under "AmPAL Devices."

Section 6: Appendices
Includes quick reference information.

If you have any questions or comments on PLDs or any other products, please contact your most convenient AMD/MMI sales office, listed at the end of each book.

Trademarks

PAL®, HAL®, PALASM®, and SKINNYDIP® are registered trademarks of Monolithic Memories, Inc.

ProPAL™, MegaPAL™, ZPAL™, MegaHAL™, ZHAL™, PLE™, PLEASM™, DOC™, Diagnostics-On-Chip™, PROSE™, MONOX™, HiPAC™, and AutoVec™ are trademarks of Monolithic Memories, Inc.

IMOX™ and SSR™ are trademarks of Advanced Micro Devices.

Xilinx™, XACT™, XACTOR™, Logic Cell Array™, and Logic Processor™ are trademarks of Xilinx Inc.

P-SILOS™ is a trademark of SimuCad.

IBM® is a registered trademark of International Business Machines Corporation.

Micro Channel™, IBM-PC™, PC-XT™, and PC-AT™ are trademarks of International Business Machines Corporation.

Data I/O® is a registered trademark of Data I/O Corporation.

ABEL™, PLDtest™, UniSite™, LogicPak™, Logic Fingerprint™, and PROMlink™ are trademarks of Data I/O Corporation.

FutureNet® is a registered trademark of FutureNet, a Data I/O Company.

DASH™, DASH-ABEL™, DASH-GATES™, PLD-CADAT™, and DASH-CADAT-PLUS™ are trademarks of FutureNet, a Data I/O Company.

CUPL™ is a trademark of Personal CAD Systems.

Multibus™ is a trademark of Intel Corporation.

ISDATA® is a registered trademark of ISDATA GmbH.
LOG/iC™ is a trademark of ISDATA GmbH.

Apollo® is a registered trademark of Apollo Computer.

Q-Bus®, UNIBUS®, VAX®, and VMS® are registered trademarks of Digital Equipment Corporation.

Microsoft® is a registered trademark of Microsoft Corporation.
MS-DOS™ is a trademark of Microsoft Corporation.

UNIX™ is a trademark of AT&T Technologies, Inc.

DAISY® and DNIX® are registered trademarks of Daisy Systems.

SOFTPACK™, SOFTLINK™, and LOGILINK™ are trademarks of Digelec, Inc.

ALLPRO™ and LOGIPRO™ are trademarks of Logical Devices, Inc.

PROMAC™ is a trademark of Japan Macnics Corporation.

WordStar™ is a trademark of MicroPro International Corporation.

Mouse System™ is a trademark of Mouse Systems Corporation

TestPLA™ is a trademark of Structured Design, Inc.

NuBus™ is a trademark of Texas Instruments

Table of Contents

PAL Device Handbook

Introduction

Introduction 1-1
- What is a PLD? 1-1
- What Advantages do PLDs Have Over Other Implementations? 1-4

Product Overview 1-7
- PAL Devices 1-7
- Nomenclature 1-7
- Programmable Sequencers 1-19
- LCA Devices 1-20

Applications

Beginner's Guide 2-1
- Constructing a Combinatorial Design 2-2
- Constructing a Registered Design 2-9
- Programming a Device 2-14

PLD Design Methodology 2-21
- Conceptualizing a Design 2-22
- Device Selection Considerations 2-23
- Implementing a Design 2-26
- Simulation 2-30
- Device Programming and Testing 2-33

Combinatorial Logic Design 2-35
- Encoders and Decoders 2-35
- Multiplexers 2-43
- Comparators 2-45
- Range Decoders 2-52
- Adders/Arithmetic Circuits 2-53
- Latches 2-58

Registered Logic Design 2-61
- Binary Counters 2-67
- Modulo Counters 2-75
- Gray-Code Counters 2-87
- Johnson Counters 2-88
- Shift Registers 2-90
- Asynchronous Registered Designs 2-94

State Machine Design 2-101
State Machine Theory 2-103
State Machine Types: Mealy & Moore 2-105
Device Selection Considerations 2-107
PAL Devices as Sequencers 2-111
Programmable Logic Sequencers (PLS) 2-117
PROSE Sequencer (PMS14R21) 2-120
Fuse Programmable Controller (Am29PL141) 2-121
State Machine Design Tutorial 2-122

Microprocessor-Based Systems 2-131
Interfacing to the 8086/80186/80286 2-134
8086 and Am7990 LANCE Interface 2-135
8086 and Am9516 Universal DMA Controller Interface 2-138
80286 to Am9568 Data Ciphering Processor Interface 2-143
80286 to Am8530 Interface 2-147
Interfacing to the 68000/68020 2-149
The 68000 and Am8530 Interface with Interrupts 2-150
68000 and Am7990 LANCE Interface 2-153
68000 to AmZ8068 Data Ciphering Processor Interface 2-155
68000 and Dual Am9516 DMA Controllers Interface 2-159
Am8530 to 68020 Interface 2-162
Interfacing to the 8088 2-165
8088 to Am9516 UDC Interface 2-166
80186 to Am9516 Universal DMA Controller Interface 2-170
68000 Interrupt Controller 2-172

Memory Control 2-179
Memory Handshake Logic 2-184
Customize a DRAM Controller Using Advanced PAL Devices 2-187
8088 to Am2968 Interface 2-202
MC68000 to Am2968 Interface 2-210
General-Purpose Dual-Port Arbiter 2-215
Dynamic Memory Control State Sequencer 2-224
8-Bit Error Detection and Correction 2-229
Fuse Programmable Controller Simplifies Cache Design 2-239
PAL22RX8A Provides Control and Addressing for a 32-Location-Deep RAM-Based LIFO 2-250

Graphics and Image Processing Systems 2-257
Small System Video Controller 2-261
PAL32VX10 Uses Buried Register for Input-Intensive State Machine Designs 2-275

Digital Signal Processing 2-283
Waveform Generator 2-286
PAL32VX10 Uses Buried Register for Input-Intensive State Machine Designs 2-292
Analog to Digital Conversion 2-297
PAL Devices, PROMs, FIFOs and Multipliers Team up to
Implement Single-Board High-Performance Audio Spectrum Analyzer 2-307

Bus Interface 2-325
Unibus Interrupt Controller 2-328
A MULTIBUS Arbiter Design for 10 MHz Processors 2-333
MULTIBUS to Am9516 Interface 2-338
Z-BUS and 8088/8086 Interface 2-342
VME Bus Control Simplified with PLDs 2-347

Table of Contents

Communications 2-357
- B8ZS Coding Using CMOS ZPAL Devices 2-362
- HDB3 Line Coding Using PAL Devices 2-384
- ZPAL Devices Implement D4 Frame Synchronization 2-404
- T1 Extended Superframe Provides Transmission Error Detection 2-431
- Time Division Multiplexing with the LCA Device 2-435
- LCA Device Implements an 8-Bit Format Converter in a PBX Switching Module 2-444
- Serial Data Link Controller 2-453
- QAM Encoder in a ZPAL Device 2-456
- PAL Devices Implement the Full V.32 Convolution Encoder 2-463
- PLD Devices Implement 4B3T Line Transcoder 2-475
- PALC22V10 Creates Manchester Encoder Circuit 2-499

Peripheral Design 2-507
- Building an ESDI Translator Using the M2064 Logic Cell Array 2-509
- Writing a Tape Drive Controller in PROSE 2-519
- GCR (4B-5B) Encoder/Decoder 2-541

Industrial Control 2-547
- Stepper Motor Controllers 2-550
- Shaft Encoders 2-558

Military Applications 2-579
- Radiation Hardness 2-581

Article Reprints
- Programmable Logic Device Preserves Pins, Product Terms (PAL32VX10) 2-589
- PLD Programmability Extends its Sway Over Complex I/O (PALC29M/MA16) 2-593
- PROSE Devices Simplify State Machine Design (PMS14R21) 2-601
- Designing a State Machine with a Programmable Sequencer (PMS14R21) 2-607
- FPCs and PLDs Simplify VME Bus Control (Am29PL141) 2-619
- Fuse-Programmable Chip Takes Command of Distributed Systems (Am29PL141) 2-633
- PAL Device Buries Registers, Brings State Machines to Life (AmPAL23S8) 2-639
- Programmable Event Generator Conquers Timing Restraints (Am2971) 2-644
- Wait-State Remover Improves System Performance (PAL22V10) 2-648
- PLDs Implement Encoder/Decoder for Disk Drives (PAL22V10) 2-651
- Mixing Data Paths Expands Options in System Design (PAL22V10) 2-660
- Programmable Logic Chip Rivals Gate Arrays in Flexibility (PAL22V10) 2-670
- XOR PLDs Simplify Design of Counters and Other Devices (PAL20X10) 2-676
- The PAL20RA10 Story —The Customization of a Standard Product (PAL20RA10) 2-683
- PLDs Abound: RAM-Based Logic Joins In 2-699
- Introduction to Programmable Array Logic 2-702
- Logical Alternatives in Supermini Design 2-711

Conference Proceedings
- New PAL Device Architecture Extends Design Flexibility (PAL32VX10) 2-719
- PROSE Architecture and Design Methodology (PMS14R21) 2-724
- Blazing Fast PAL Devices Enable New Application Areas (PAL16R8-10, PAL10H20P/G8) 2-730

Sales Offices 2-740

PAL Device Data Book

Programming and Quality

Design Software for Programmable Logic 3-1
- PALASM 2 Logic Design Software Package 3-5
- PLPL: Programmable Logic Programming Language 3-7
- Logic Cell Array and Development Systems 3-21
- ABEL-GATES 3-35
- CUPL 3-43
- LOG/iC 3-66

Programming 3-76
- Programmer Reference Guide 3-81
- ProPAL, HAL and ZHAL Devices Program 3-104

Testability 3-108
- Designing Testable Combinatorial Circuits 3-109
- Designing Testable Sequential Circuits 3-114
- Designing Testable State Machines 3-118
- Designing for Testability with the PROSE Device 3-123
- Using Test Vectors 3-128

Technology and Quality
- MONOX 3 Oxide-Isolated Process 3-130
- Product Assurance 3-132
- Test and Finish Operations 3-138
- IMOX Product Technology and Reliability 3-140
  - IMOX Product Reliability 3-141
  - IMOX Product Testability 3-144
- ECL Technology 3-150
- CMOS HiPAC Technology 3-155
- CMOS EE Technology 3-159
- Latchup in CMOS Integrated Circuits 3-160
- Metastability 3-164

Packaging
- Surface Mount Technology 3-170
- PAL Device Package Outlines 3-179
- PAL Device Package Thermal Characteristics 3-208
- AmPAL Device Package Outlines 3-224
- AmPAL Device Package Thermal Characteristics 3-231

PALASM 2 Software User Documentation

Introduction 4-1
Install PALASM 2 Software 4-15
Run the Software 4-29
Build a Boolean Equation Design 4-61
Build a State Machine Design 4-95
Build Simulation 4-137
Program the Device 4-169
PALASM 2 Software Glossary 4-183
PALASM 2 Software Index 4-189

Data Sheets

PAL/PLD Device Menu 5-3

TTL/CMOS PAL Devices 5-9

PAL16RA8 5-11

PAL16RP8A Series 5-17
- PAL16P8A
- PAL16RP8A
- PAL16RP6A
- PAL16RP4A

PAL16R8 Family 5-26

PAL16R8D Series 5-29
- PAL16L8D
- PAL16R8D
- PAL16R6D
- PAL16R4D

PAL16R8B Series 5-31
- PAL16L8B
- PAL16R8B
- PAL16R6B
- PAL16R4B

PALC16R8Q-25 Series 5-33
- PAL16L8Q-25
- PAL16R8Q-25
- PAL16R6Q-25
- PAL16R4Q-25

PAL16R8B-2 Series 5-35
- PAL16L8B-2
- PAL16R8B-2
- PAL16R6B-2
- PAL16R4B-2

PAL16R8A Series 5-37
- PAL16L8A
- PAL16R8A
- PAL16R6A
- PAL16R4A

PAL16R8B-4 Series 5-39
- PAL16L8B-4
- PAL16R8B-4
- PAL16R6B-4
- PAL16R4B-4

PAL16R8A-2 Series 5-41
- PAL16L8A-2
- PAL16R8A-2
- PAL16R6A-2
- PAL16R4A-2

PAL16R8A-4 Series 5-43
- PAL16L8A-4
- PAL16R8A-4
- PAL16R6A-4
- PAL16R4A-4

PALC16R8Z-25 Series 5-50
- PAL16L8Z-25
- PAL16R8Z-25
- PAL16R6Z-25
- PAL16R4Z-25

PAL16X4 5-51

PAL10H8 Series 5-56
- PAL10H8
- PAL12H6
- PAL14H4
- PAL16H2
- PAL16C1
- PAL10L8
- PAL12L6
- PAL14L4
- PAL16L2

PAL32VX10A 5-70

PAL32VX10 5-70

PALC22V10H-25 5-79

PALC22V10H-35 5-79

PAL22RX8A 5-87

PAL20RA10-20 5-95

PAL20RA10 5-97

PAL20RS10 Series 5-103
- PAL20S10
- PAL20RS10
- PAL20RS8
- PAL20RS4

PAL20X10A Series 5-113
- PAL20L10A
- PAL20X10A
- PAL20X8A
- PAL20X4A

PAL20R8 Family 5-122
PAL20R8B Series 5-125
PAL20L8B
PAL20R8B
PAL20R6B
PAL20R4B
PAL20R8B-2 Series 5-126
PAL20L8B-2
PAL20R8B-2
PAL20R6B-2
PAL20R4B-2
PAL20R8A Series 5-128
PAL20L8A
PAL20R8A
PAL20R6A
PAL20R4A
PAL20R8A-2 Series 5-130
PAL20L8A-2
PAL20R8A-2
PAL20R6A-2
PAL20R4A-2
PALC20R8Z-35 Series 5-133
PALC20L8Z-35
PALC20R8Z-35
PALC20R6Z-35
PALC20R4Z-35
PALC20R8Z-45 Series 5-133
PALC20L8Z-45
PALC20R8Z-45
PALC20R6Z-45
PALC20R4Z-45

PAL6L16A 5-141
PAL8L14A 5-141

PAL12L10 Series 5-147
PAL12L10
PAL14L8
PAL16L6
PAL18L4
PAL20L2
PAL20C1

PAL32R16 5-158

General Information 5-164

TTL/CMOS AmPAL Devices 5-167

AmPAL23S8-20 5-169
AmPAL23S8-25 5-169

AmPAL16R8 Family 5-184
AmPAL16R8D Series 5-183
AmPAL16L8D
AmPAL16R8D
AmPAL16R6D
AmPAL16R4D
AmPAL16R8B Series 5-197
AmPAL16L8B
AmPAL16R8B
AmPAL16R6B
AmPAL16R4B
AmPAL16R8AL Series 5-197
AmPAL16L8AL
AmPAL16R8AL
AmPAL16R6AL
AmPAL16R4AL
AmPAL16R8A Series 5-197
AmPAL16L8A
AmPAL16R8A
AmPAL16R6A
AmPAL16R4A
AmPAL16R8Q Series 5-197
AmPAL16L8Q
AmPAL16R8Q
AmPAL16R6Q
AmPAL16R4Q
AmPAL16R8L Series 5-197
AmPAL16L8L
AmPAL16R8L
AmPAL16R6L
AmPAL16R4L
AmPAL16R8 Series 5-197
AmPAL16L8
AmPAL16R8
AmPAL16R6
AmPAL16R4

AmPAL18P8B 5-202
AmPAL18P8AL 5-202
AmPAL18P8A 5-202
AmPAL18P8Q 5-202
AmPAL18P8L 5-202

AmPALC29MA16-35 5-209
AmPALC29MA16-45 5-209
AmPALC29M16-35 5-231
AmPALC29M16-45 5-231

AmPAL22V10-15 5-249
AmPAL22V10A 5-260
AmPAL22V10 5-260

AmPAL20XRP10 Family 5-271
AmPAL20XRP10-20 Series 5-286
AmPAL22XP10-20
AmPAL20XRP10-20
AmPAL20XRP8-20
AmPAL20XRP6-20
AmPAL20XRP4-20
AmPAL20XRP10-30L Series 5-286
AmPAL22XP10-30L
AmPAL20XRP10-30L
AmPAL20XRP8-30L
AmPAL20XRP6-30L
AmPAL20XRP4-30L
AmPAL20XRP10-30 Series 5-286
AmPAL22XP10-30
AmPAL20XRP10-30
AmPAL20XRP8-30
AmPAL20XRP6-30
AmPAL20XRP4-30
AmPAL20XRP10-40L Series 5-286
AmPAL22XP10-40L
AmPAL20XRP10-40L
AmPAL20XRP8-40L
AmPAL20XRP6-40L
AmPAL20XRP4-40L

AmPAL20RP10 Family 5-291
AmPAL20RP10B Series 5-306
AmPAL22P10B
AmPAL20RP10B
AmPAL20RP8B
AmPAL20RP6B
AmPAL20RP4B
AmPAL20RP10AL Series 5-306
AmPAL22P10AL
AmPAL20RP10AL
AmPAL20RP8AL
AmPAL20RP6AL
AmPAL20RP4AL
AmPAL20RP10A Series 5-306
AmPAL22P10A
AmPAL20RP10A
AmPAL20RP8A
AmPAL20RP6A
AmPAL20RP4A

AmPAL20L10B 5-306
AmPAL20L10-20 5-306
AmPAL20L10AL 5-306

PROSE/PLS Sequencers 5-313
PMS14R21A 5-315
PMS14R21 5-315

PLS167-33 5-331
PLS168-33 5-331
PLS105-37 5-331

FPC/PEG Sequencers 5-337
Am29PL141 Fuse Programmable Controller 5-339
Am2971 Programmable Event Generator 5-365

ECL PAL Devices 5-379
PAL10020EV/EG8 5-381
PAL10H20EV/EG8 5-381
PAL10H20G8 5-382
PAL10H20P8 5-386

HAL/ZHAL Devices 5-391
ZHAL20A Series 5-394
ZHAL24A Series 5-401

Military PAL Devices 5-415
Introduction 5-417
Military PAL/PLD Device Menu 5-418
Military 20-pin PAL Devices 5-421
Military 24-pin PAL Devices 5-439
DC/AC Parametric Testing 5-469
JAN 38510 and Standard Military Drawings 5-470
Military Screening 5-474
Quality Programs 5-477

Logic Cell Array 5-481
M2064 5-483
M2018 5-483
Military M2064/M2018 5-518

Electrical Definitions 5-531

Appendices

Logic Reference 6-1
- Basic Logic Elements 6-1
- Basic Storage Elements 6-8
- Binary Numbers 6-15

Signal Polarity 6-19
Glossary 6-24
Competitive Cross-Reference 6-30
Worldwide Application Support 6-40
Sales Offices 6-41

Alphanumeric Product Index

Am29PL141 5-339
Am2971 5-365
AmPAL16L8 5-197
AmPAL16L8A 5-197
AmPAL16L8AL 5-197
AmPAL16L8B 5-197
AmPAL16L8D 5-183
AmPAL16L8L 5-197
AmPAL16L8Q 5-197
AmPAL16R4 5-197
AmPAL16R4A 5-197
AmPAL16R4AL 5-197
AmPAL16R4B 5-197
AmPAL16R4D 5-183
AmPAL16R4L 5-197
AmPAL16R4Q 5-197
AmPAL16R6 5-197
AmPAL16R6A 5-197
AmPAL16R6AL 5-197
AmPAL16R6B 5-197
AmPAL16R6D 5-183
AmPAL16R6L 5-197
AmPAL16R6Q 5-197
AmPAL16R8 5-197
AmPAL16R8A 5-197
AmPAL16R8AL 5-197
AmPAL16R8B 5-197
AmPAL16R8D 5-183
AmPAL16R8L 5-197
AmPAL16R8Q 5-197

AmPAL18P8A 5-202
AmPAL18P8AL 5-202
AmPAL18P8B 5-202
AmPAL18P8L 5-202
AmPAL18P8Q 5-202

AmPAL20L10AL 5-306
AmPAL20L10B 5-306
AmPAL20L10-20 5-306
AmPAL20RP4A 5-306
AmPAL20RP4AL 5-306
AmPAL20RP4B 5-306
AmPAL20RP6A 5-306
AmPAL20RP6AL 5-306
AmPAL20RP6B 5-306
AmPAL20RP8A 5-306
AmPAL20RP8AL 5-306
AmPAL20RP8B 5-306
AmPAL20RP10A 5-306
AmPAL20RP10AL 5-306
AmPAL20RP10B 5-306
AmPAL20XRP4-20 5-286
AmPAL20XRP4-30L 5-286
AmPAL20XRP4-30 5-286
AmPAL20XRP4-40L 5-286
AmPAL20XRP6-20 5-286
AmPAL20XRP6-30L 5-286
AmPAL20XRP6-30 5-286
AmPAL20XRP6-40L 5-286
AmPAL20XRP8-20 5-286
AmPAL20XRP8-30L 5-286
AmPAL20XRP8-30 5-286
AmPAL20XRP8-40L 5-286
AmPAL20XRP10-20 5-286
AmPAL20XRP10-30L 5-286
AmPAL20XRP10-30 5-286
AmPAL20XRP10-40L 5-286

AmPAL22P10A 5-306
AmPAL22P10AL 5-306
AmPAL22P10B 5-306
AmPAL22V10 5-260
AmPAL22V10-15 5-249
AmPAL22V10A 5-260
AmPAL22XP10-20 5-286
AmPAL22XP10-30L 5-286
AmPAL22XP10-30 5-286
AmPAL22XP10-40L 5-286

AmPAL23S8-20 5-169
AmPAL23S8-25 5-169

AmPALC29M16-35 5-231
AmPALC29M16-45 5-231
AmPALC29MA16-35 5-210
AmPALC29MA16-45 5-210

M2018 5-483
M2064 5-483

PAL6L16A 5-141
PAL8L14A 5-141

PAL10H8 5-56
PAL10H20G8 5-382
PAL10H20EV/EG8 5-381
PAL10H20P8 5-386
PAL10L8 5-56

PAL12H6 5-56
PAL12L6 5-56
PAL12L10 5-147

PAL14H4 5-56
PAL14L4 5-56
PAL14L8 5-147

PAL16C1 5-56
PAL16H2 5-56
PAL16L2 5-56
PAL16L6 5-147
PAL16L8D 5-29
PAL16L8A 5-37
PAL16L8A-2 5-41
PAL16L8A-4 5-43
PAL16L8B 5-31
PAL16L8B-2 5-35
PAL16L8B-4 5-39
PAL16P8A 5-17
PAL16R4D 5-29
PAL16R4A 5-37
PAL16R4A-2 5-41
PAL16R4A-4 5-43
PAL16R4B 5-31
PAL16R4B-2 5-35
PAL16R4B-4 5-39
PAL16R6D 5-29
PAL16R6A 5-37
PAL16R6A-2 5-41
PAL16R6A-4 5-43
PAL16R6B 5-31
PAL16R6B-2 5-35
PAL16R6B-4 5-39
PAL16R8D 5-29
PAL16R8A 5-37
PAL16R8A-2 5-41
PAL16R8A-4 5-43
PAL16R8B 5-31
PAL16R8B-2 5-35
PAL16R8B-4 5-39
PAL16RA8 5-11
PAL16RP4A 5-17
PAL16RP6A 5-17
PAL16RP8A 5-17
PAL16X4 5-51

PAL18L4 5-147

PAL20C1 5-147
PAL20L2 5-147
PAL20L10A 5-113
PAL20L8A 5-128
PAL20L8A-2 5-130
PAL20L8B 5-125
PAL20L8B-2 5-126
PAL20R4A 5-128
PAL20R4A-2 5-130
PAL20R4B 5-125
PAL20R4B-2 5-126
PAL20R6A 5-128
PAL20R6A-2 5-130
PAL20R6B 5-125
PAL20R6B-2 5-126
PAL20R8A 5-128
PAL20R8A-2 5-130
PAL20R8B 5-125
PAL20R8B-2 5-126
PAL20RA10 5-97
PAL20RA10-20 5-95
PAL20RS10 5-103
PAL20RS4 5-103
PAL20RS8 5-103
PAL20S10 5-103
PAL20X4A 5-113
PAL20X8A 5-113
PAL20X10A 5-113

PAL22RX8A 5-87

PAL32R16 5-158
PAL32VX10 5-70
PAL32VX10A 5-70

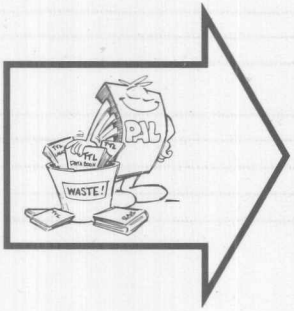
PAL10020EV/EG8 5-381

PALC16L8Q-25 5-33
PALC16L8Z-25 5-50
PALC16R4Q-25 5-33
PALC16R4Z-25 5-50
PALC16R6Q-25 5-33
PALC16R6Z-25 5-50
PALC16R8Q-25 5-33
PALC16R8Z-25 5-50
PALC20L8Z-35 5-133
PALC20R4Z-35 5-133
PALC20R6Z-35 5-133
PALC20R8Z-35 5-133
PALC20L8Z-45 5-133
PALC20R4Z-45 5-133
PALC20R6Z-45 5-133
PALC20R8Z-45 5-133
PALC22V10H-25 5-79
PALC22V10H-35 5-79

PLS105-37 5-331
PLS167-33 5-331
PLS168-33 5-331

PMS14R21 5-315
PMS14R21A 5-315

IMPORTANT
NOTES



PAL Device Handbook

Introduction 1
Applications 2

PAL Device Data Book

Programming and Quality 3
PALASM 2 Software User Documentation 4
Data Sheets 5
Appendices 6

Table of Contents

Introduction 1-1
- What is a PLD? 1-1
- What Advantages do PLDs Have Over Other Implementations? 1-4

Product Overview 1-7
- PAL Devices 1-7
- Nomenclature 1-7
- Programmable Sequencers 1-19
- LCA Devices 1-20

Introduction

The Programmable Array Logic device, commonly known as the PAL® device, was invented at Monolithic Memories over 12 years ago. The concept for this revolutionary type of device sprang forth as a simple solution to the shortcomings of discrete TTL logic.

The successfully proven PROM technology which allowed the end user to "write on silicon" provided the technological basis which made this kind of device not only possible, but very popular as well.

The availability of design software made it much easier to design with programmable logic. As designers were freed from the drudgery of low-level implementation issues, new complex designs were easier to implement, and could be completed more quickly.

This chapter outlines some basic information essential to those who are unfamiliar with Programmable Logic devices (PLDs). The information may also be useful to those who are current users of programmable logic. The specific issues which need to be addressed are:

- What is a PLD?
- What other implementations are possible?
- What advantages do PLDs have over other implementations?

What is a PLD?

In general, a programmable logic device is a circuit which can be configured by the user to perform a logic function. Most "standard" PLDs consist of an AND array followed by an OR array, either (or both) of which is programmable. Inputs are fed into the AND array, which performs the desired AND functions and generates product terms. The product terms are then fed into the OR array. In the OR array, the outputs of the various product terms are combined to produce the desired outputs. There are three fundamental types of standard PLD: the PROM, the PAL device, and the PLS device.

PROMs

PROMs are usually thought of as memory elements. However, the PROM has a fixed AND array (which decodes the memory address) followed by a programmable OR array (Figure 1). For each of a given set of input combinations (addresses), it generates a value which has been programmed into the device.

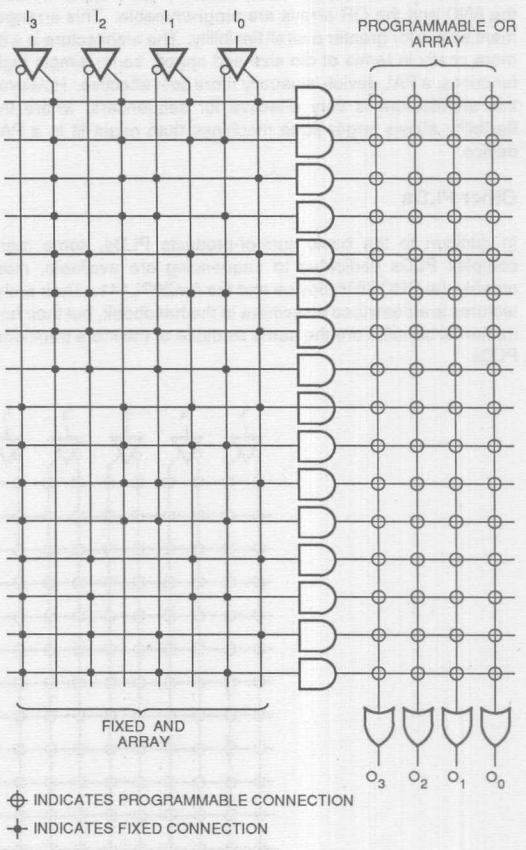


Figure 1. PROM Array Structure

PAL Devices

The PAL device has a programmable AND array followed by a fixed OR array (Figure 2). The fact that the AND array is programmable makes it possible for the devices to have many inputs. The fact that the OR array is fixed makes the devices small (which means less expensive) and fast.

PLS Devices

The PLS (Programmable Logic-based sequencer) devices are based on the standard PLA architecture (Figure 3), where both the AND and the OR arrays are programmable. This arrangement allows for greater overall flexibility. The architecture is a bit more costly in terms of die size and speed, so for simple logic functions, a PAL device is usually more cost effective. However, this architecture is very effective for sequencers, where the flexibility allows larger state machines than might fit in a PAL device.

Other PLDs

In addition to the basic sum-of-products PLDs, some more complex PLDs dedicated to sequencing are available, most notably the PROSE™ device and the Am29PL141. Their architectures are described elsewhere in the handbook, but their fundamental benefits are the same as those of the more traditional PLDs.

In practice, the distinctions between architectures are not as significant as the differences between the types of functions to be performed. For this reason, this handbook is organized primarily into discussions about PAL devices and discussions about sequencers, whether those sequencers be implemented in PAL devices, PLS devices, or one of the dedicated sequencers.

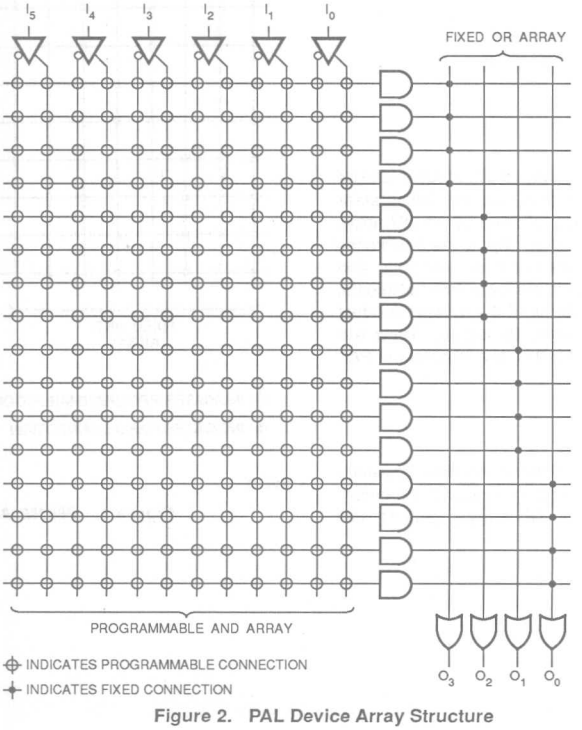
What Other Implementations Are Possible?

There are essentially four alternatives to programmable logic:

- Discrete Logic
- Gate Arrays
- Standard Cell Circuits
- Full Custom Circuits

Discrete Logic

Discrete logic, or conventional TTL logic, has the advantage of familiarity; hence its popularity. It is also quite inexpensive when only unit cost is considered. The drawback is that the implementation of even a simple portion of a system may require many units of discrete logic. There are "hidden" costs associated with each unit that goes into a system, which can render the overall system more expensive.



401 02

Figure 2. PAL Device Array Structure

Designing with discrete chips can also be very tedious. Each design decision directly affects the layout of the board. Changes are difficult to make. The design is also more difficult to document, making it harder to debug and maintain later. These items all contribute to a long design cycle when discrete chips are used extensively.

Gate Arrays

Gate arrays have been increasing in popularity. The attractiveness of this solution lies in the device's flexibility. By packing the functions into the device, a great majority of the available silicon is actually used. Since such a device is customized for an application, it would seem to be the optimum device for that application.

However, one also pays substantial development costs, especially in the case of a design which needs changes after silicon has already been processed. Even though the unit costs are generally quite low for gate arrays, the volumes required to make their use worthwhile excludes them as a solution for many designers. This fact, added to the long design cycle and high risk involved, make this solution practical for only a limited number of designers.

Standard Cell Circuits

Standard cell circuits are quite similar to gate arrays, their main advantage being that they consist of a collection of different parts of circuits which have already been debugged. These circuits are then assembled and collected to perform the desired functions. This can ideally lead to reduced turnaround from conception to implementation, and a much more efficient circuit.

The drawback is that even though the individual components of the circuit have been laid out, a complete layout must still be performed to arrange the cells. Instead of just customizing the metal interconnections, as is done in a gate array, the circuit must be developed from the bottom up. Development costs can be even higher than for gate arrays, and despite the standard cell concept, turnaround time often tends to be longer than planned. Again, the volume must be sufficiently high to warrant the development costs.

Full Custom Circuits

Full custom designs require that a specific chip be designed from scratch to perform the needed functions. The intent is to provide a solution which gives the designer exactly what is needed for the application in question; no more and no less. Ideally, not a square micron of silicon is wasted. This normally results in the smallest piece of silicon possible to fit the needs of the design, which in turn reduces the system cost. Understandably, though, development costs and risks for such a design are extremely high, and volumes must be commensurately high in order for such a solution to be of value.

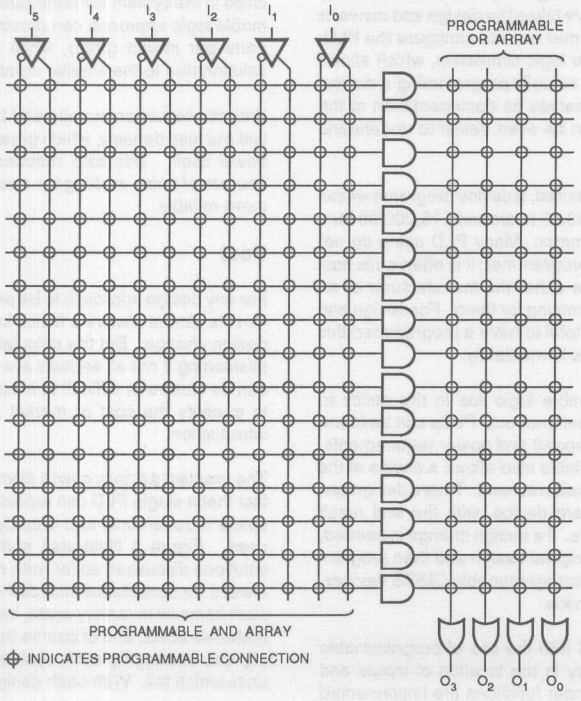


Figure 3. PLA Array Architecture

What Advantages do PLDs Have Over Other Implementations?

As user-programmable semicustom circuits, PLDs provide a valuable compromise which combines many of the benefits of discrete logic with many of the benefits of other semicustom circuits. The overall advantages can be found in several areas:

- Ease of design
- Performance
- Reliability
- Cost savings

Ease of Design

The support tools available for use in designing with PLDs greatly simplify the design process by making the lower-level implementation details transparent. In a matter of one or two hours, a first time PLD user can learn to design with a PAL device, program it, and implement the design in a system.

The design support tools consist of design software and a programmer. The design software is used in generating the design; the programmer is used to configure the device. The software provides the link between the higher-level design and the low-level programming details.

All of the available design software packages (of which Monolithic Memories' PALASM® software is the most widely used) perform essentially the same tasks. The design is specified with relatively high-level constructs; the software takes the design and converts it into a form which the programmer uses to configure the PLD. Most software packages provide logic simulation, which allows one to debug the design before actually programming a device. The high-level design file also serves as documentation of the design. This documentation can be even easier to understand than traditional schematics.

Depending on the capabilities desired, a device programmer can cost anywhere from under $1,000.00 to around $15,000.00 for a high-volume production programmer. Many PLD users do not find it necessary to purchase a programmer; it is often quite cost effective and convenient to have either the manufacturer or an outside distributor do the programming for them. For design and prototyping, though, it is very helpful to have a programmer; this allows one to implement designs immediately.

The convenience of programmable logic lies in the ability to customize a standard, off-the-shelf product. PLDs can be found in stock to suit a wide range of speed and power requirements. The variety of architectures available also allows a choice of the proper functionality for the application at hand. Thus a design can be implemented using a standard device, with the end result essentially being a custom device. If a design change is needed, it is a simple matter to edit the original design and then program a new device, or, in the case of reprogrammable CMOS devices, erase and reprogram the old device.

Board layout is vastly simplified with the use of programmable logic. PLDs offer great flexibility in the location of inputs and outputs on the device. Since larger functions are implemented inside the PLD, board layout can begin once the inputs and outputs are known. The details of what will actually be inside the PLD can be worked out independently of the layout. In many cases, any needed design changes can be taken care of entirely within the PLD, and will not affect the PC board.

Performance

Speed is one of the main reasons that designers use PAL devices. The TTL PAL devices presently on the market can provide equal or better performance than the fastest discrete logic available. ECL PAL devices extend the benefits of programmable logic to the even higher-speed realm of ECL logic. Today's fastest PAL devices are being developed on the newest technologies to gain every extra nanosecond of performance.

Performance cannot come strictly at the expense of power consumption. Since PLDs can be used to replace several discrete circuits, the power consumption of a PLD may well be less than that of the combined discrete devices. As more PLDs are developed in CMOS technology, the option for even lower power becomes available, including zero standby power devices for systems which can tolerate only minute standby power consumption.

Reliability

Reliability is an area of increasing concern. As systems get larger and more complex, the increase in the amount of circuitry tends to reduce the reliability of the system; there are "more things to go wrong". Thus a solution which inherently reduces the number of chips in the system will contribute to higher reliability. A programmable logic approach can provide device quality levels up to 50 parts per million (ppm), while also providing a more reliable solution due to the smaller number of devices required.

With the reduction in units and board space, PC boards can be laid out less densely, which greatly improves the reliability of the board itself. This also reduces crosstalk and other potential sources of noise, making the operation of the system cleaner and more reliable.

Cost

For any design approach to be practical, it must be cost effective. Cost is almost always a factor in considering a new design or a design change. But the calculation of total system cost can be misleading if not all aspects are considered. Many of the costs can be elusive or difficult to measure. For example, it is difficult to quantify the cost of market share lost due to late product introduction.

The greatest savings over a discrete design are derived from the fact that a single PLD can replace several discrete chips. Board space requirements can drop by 25% or more when PLDs are used. Figure 4 illustrates some of the costs of the various solutions discussed so far, with many of the factors that may not always be considered included for comparison. These involve such items as inventory costs, inspection costs, test costs, board materials costs, and of course the very costly time spent designing and debugging such systems, and isolating and replacing units which fail. With each design change, the cost of a custom

	Unit Cost Range	NREs/ Development Costs	Purchase Volume For Best Economics	Design Flexibility	Gate Count per Device	Engineering Design Time	Design Turnaround Time	Cost of Each Design Change
PLDs: BLANK	Low–Med	None	5K–200K	Med	200–2K	1/2 Week	Short (1–10 Days)	Very Low
PLDs: FACTORY PROGRAMMED	Low–Med	Low	10K–200K	Med	200–2K	3–10 Weeks	Moderate (8–10 Wks.)	Med
DISCRETE LOGIC	Low–Med	None	1K–10K	Low	10–30	1 Week	Short (1–10 Days)	Med
GATE ARRAYS	Low	Med–High	10K–200K	Med–High	1K–10K	12–40 Weeks	Long (3–9 Mos.)	High
STANDARD CELLS	Low	High	100K–300K	Med–High	1K–10K	26–52 Weeks	Long (6–12 Mos.)	High
FULL CUSTOM	Very Low	High	200K and Up	High	1K–60K	1–2 Yrs.	Long (6–12 Mos.)	High

Figure 4. Cost and Design-time Comparisons

solution rises dramatically, while that of a user-customizable approach is minimal. The relationship between the various alternatives is summarized in Figure 5.

Another economic benefit of the use of PLDs is that when one PAL device is used in several different designs, as is often the case, the user has not committed that device to any one of the particular designs until the device has been programmed. This means that inventory can be stocked for several different designs in the form of one device. As requirements change, the parts can be programmed to fit the need. And in the case of reprogrammable CMOS devices, one is not committed even after programming.

There is also a cost-effective PLD solution for high-volume production. Just as a ROM is a PROM which has been hard-wired for mass production, HAL® (Hard Array Logic) devices can be produced in high volumes for extra cost savings. This mask-programmed version can be produced in high volume with unparalleled quality. In addition, in the event that production quantities for your system show an unexpected increase, the equivalent PAL or ProPAL™ devices (in-factory programmed PAL devices) can be quickly obtained and programmed in your factory, by Monolithic Memories, or through distribution to cover the temporary shortage. More details on HAL and ProPAL devices are provided on page 3-104 of the Data Book.

One final subtle cost issue is derived from the ease with which a competitor can copy a design. PLDs have a unique feature called a security fuse, whose purpose is to protect a design from being copied. By using secured PLDs extensively in a system, one can safely avoid having one's system easily deciphered. The added design security provided by this feature can buy extra market time, forcing competitors to do their own original design work rather than copying the designs of others.

Summary

Programmable logic provides the means of creating semi-custom designs with readily available standard components. There is a wide variety of PLDs; PAL devices are most widely used, and perform well for basic logic and some sequencing functions. Other dedicated sequencers provide the circuitry required to implement more complex designs.

By assuming some of the attributes of gate arrays, programmable logic provides the cost savings of any other semicustom device, without the extra engineering costs, risks, and design delays. Reliability is also enhanced as quality increases and board complexity decreases.

The design tasks are greatly simplified due to the design tools which are now available. Design software and device programmers allow top-down high-level designs, with a minimum of time spent on actual implementation issues. Simulation allows some design debug before a device is programmed.

For all of these reasons, programmable logic has become, and will continue to be, the design methodology of choice among digital systems designers.

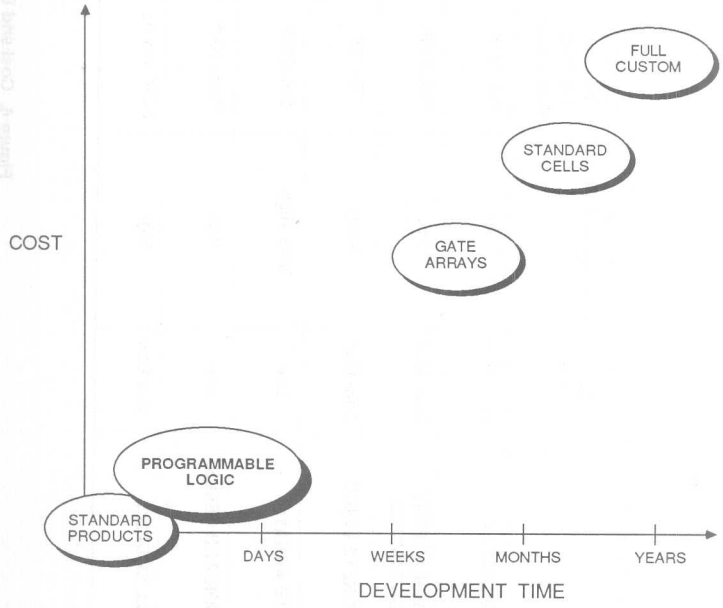


Figure 5. Development Cost vs Time for Alternative Logic Implementations

Product Overview

Introduction

Monolithic Memories and Advanced Micro Devices offer a wide variety of PLDs, implemented in a variety of technologies. In this section, we will briefly discuss the device families, and look at the various architecture, speed, and power options. More specific device information can be found in the individual data sheets in Section 5. Discussions on some of the special architectural features of many of the devices can also be found in their respective datasheets.

There are three basic PLD areas addressed by Monolithic Memories' devices:

- PAL devices—general purpose
- Programmable sequencers—state machines
- LCA™ devices—high density

The largest application area is that covered by the PAL devices. There is a wide variety of PAL devices, ranging from simple devices that address general logic design problems to more sophisticated devices that deal with more complex problems.

There is also a series of sequencers which are not PAL devices, featuring architectures particularly well suited to sequencing operations. While there are PAL devices that work well as state machines in addition to their other applications, these dedicated sequencers have given up some of their generality to provide optimal state machine solutions.

The final area covered is that of high-density design, addressed by the LCA devices. The LCA device takes the approach of a programmable gate array to provide a PLD with many usable gates.

Design software radically simplifies the design of any circuit in a PLD. PALASM software is used for all of the PLDs except the LCA devices. The LCA devices make use of XACT™ software for logic configuration. The PALASM software documentation is in Section 4; XACT software is discussed more fully on page 3-17 of the Data Book.

PAL Devices

PAL devices are available in three different technologies:

- TTL
- CMOS
- ECL

The CMOS devices often provide the same functions as the TTL devices, and can be used in the same sockets, with the added benefit of lower power. Thus the TTL and CMOS devices will be discussed together, followed by a summary of specific CMOS issues. ECL devices require completely different design considerations, and cannot be interchanged with TTL or CMOS devices. They will therefore be discussed separately.

PAL devices generally have a mnemonic naming scheme, which provides some basic information about the devices' capabilities.

Because Monolithic Memories and Advanced Micro Devices recently merged, there are presently two slightly different naming conventions in use. Some devices were originally Monolithic Memories' devices; some were Advanced Micro Devices'; some devices were produced by both companies before the merger. In order to minimize the confusion after the merger, the old device names are being maintained. The two naming conventions will be discussed separately. Throughout this handbook, "AmPAL" designates products originally from Advanced Micro Devices.

Note that all products that Monolithic Memories and Advanced Micro Devices were producing before the merger are still in production.

Monolithic Memories Nomenclature

Monolithic Memories originally set the PAL device naming standard. An example of an older device name is shown in Figure 1.

As devices have become faster, and as CMOS has become available, it has become necessary to change the way we describe the performance of newer devices. While the convention is slightly different, it does make it easier to see exactly what the speed grade and power level of a device are. An example of such a new device is shown in Figure 2.

Advanced Micro Devices Nomenclature

The Advanced Micro Devices nomenclature follows the Monolithic Memories nomenclature with a few modifications. A typical older device name is shown in Figure 3.

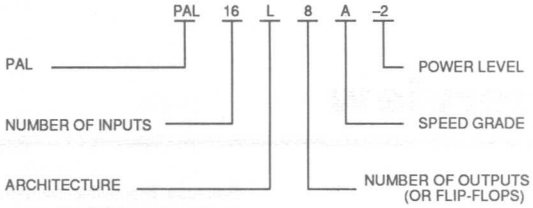
Advanced Micro Devices also found the need to modify the nomenclature with the advent of faster devices and CMOS devices. An example of a newer device is shown in Figure 4.

The relationship between a device and its name will become apparent in the Tables following.

The package designators also differ. A cross reference for commercial packages is shown in Table 1.

PACKAGE	MONOLITHIC MEMORIES	ADVANCED MICRO DEVICES
Plastic DIP	N	P
Plastic SKINNYDIP®	NS	P
Ceramic DIP	J	D
Ceramic SKINNYDIP	JS	D
PLCC–20-pin	NL	J
PLCC from 24-pin DIP, JEDEC	FN	J
PLCC from 24-pin DIP	NL	–
PLCC from 28-pin DIP	FN	J
LCC	L	L

Table 1. Package Designators



Notes:

ECL devices also have either '10H' or '100' following 'PAL' to indicate the ECL family compatibility.

The number of inputs includes feedback, if feedback exists.

The architecture code varies. It is usually mnemonic, and the meanings of each code will be listed in each of the following sections.

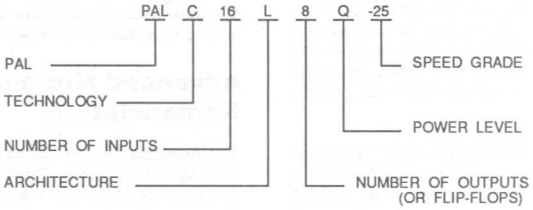
The speed grade indicates the propagation delay. In general, the speeds are as follows:

blank: 35 ns (Note that –2 and –4 devices are slower)
A: 25 ns
B: 15 ns
D: 10 ns

The power level can be one of three designators:

blank: 'full power', 180–240 mA
–2: 'half power', 90–105 mA
–4: 'quarter power', 45–55 mA

Figure 1. Monolithic Memories Nomenclature for Older Products



Notes:

The technology designator can be one of four codes:

blank = TTL
C = CMOS
10H = 10KH ECL
100 = 100K ECL

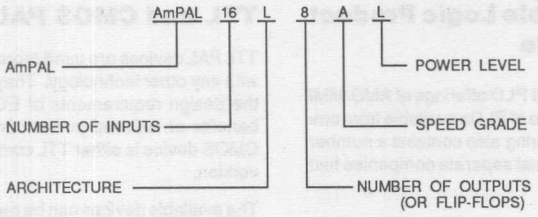
The number of inputs includes feedback, if feedback exists.

The architecture code varies. It is usually mnemonic, and the meanings of each code will be listed in each of the following sections.

The power level can be one of four designators:

blank: 'full power', 180–240 mA
H: 'half power', 90–105 mA
Q: 'quarter power', 45–55 mA
Z: 'zero power', <0.1 mA standby current

Figure 2. New Monolithic Memories Nomenclature



402 03

Notes:

The number of inputs includes feedback, if feedback exists.

The architecture code varies. It is usually mnemonic, and the meanings of each code will be listed in each of the following sections.

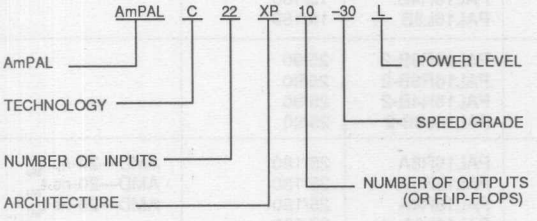
The speed grade indicates the propagation delay. In general, the speeds are as follows:

blank: 35 ns
A: 25 ns
B: 15 ns
D: 10 ns

The power level can be one of three designators:

blank: 'full power', 180–240 mA
L: 'half power', 90–105 mA
Q: 'quarter power', 45–55 mA

Figure 3. Advanced Micro Devices Nomenclature for Older Products



402 04

Notes:

The technology designator can be one of two codes:

blank = TTL
C = CMOS

The number of inputs includes feedback, if feedback exists.

The architecture code varies. It is usually mnemonic, and the meanings of each code will be listed in each of the following sections.

The speed grade indicates the propagation delay in nanoseconds.

The power level can be one of three designators:

blank: 'full power', 180–240 mA
L: 'half power', 90–105 mA

Figure 4. New Advanced Micro Devices Nomenclature

AMD/MMI Programmable Logic Product Cross Reference Guide

As a merged company, the combined PLD offerings of AMD/MMI provide the PLD user the broadest line of PLDs available from one supplier. The combined product offering also contains a number of devices for which each of the original separate companies had equivalent devices.

This is a guide to the devices that are equivalent and can be directly substituted (with exceptions as noted in the comments section).

These are the only products which are direct substitutes for each other, or which we will be encouraging customers to use as alternate parts.

TTL and CMOS PAL Devices

TTL PAL devices are used more widely by far than devices made with any other technology. They provide very high speed without the design requirements of ECL. CMOS devices provide the benefits of reduced power consumption and erasability. Each CMOS device is either TTL compatible, or has a TTL-compatible version.

The available devices can be generally placed into four functional groups:

- Simple combinatorial
- Simple registered
- Sequencer
- Asynchronous

AMD		MMI		COMMENTS
Part Number	Speed/Power	Part Number	Speed/Power	
AmPAL16R8D	10/180	PAL16R8D	10/180	
AmPAL16R6D	10/180	PAL16R6D	10/180	
AmPAL16R4D	10/180	PAL16R4D	10/180	
AmPAL16L8D	10/180	PAL16L8D	10/180	
AmPAL16R8B	15/180	PAL16R8B	15/180	
AmPAL16R6B	15/180	PAL16R6B	15/180	
AmPAL16R4B	15/180	PAL16R4B	15/180	
AmPAL16L8B	15/180	PAL16L8B	15/180	
AmPAL16R8AL	25/90	PAL16R8B-2	25/90	
AmPAL16R6AL	25/90	PAL16R6B-2	25/90	
AmPAL16R4AL	25/90	PAL16R4B-2	25/90	
AmPAL16L8AL	25/90	PAL16L8B-2	25/90	
AmPAL16R8A	25/180	PAL16R8A	25/180	AMD—20-ns t_{su}
AmPAL16R6A	25/180	PAL16R6A	25/180	AMD—20-ns t_{su}
AmPAL16R4A	25/180	PAL16R4A	25/180	AMD—20-ns t_{su}
AmPAL16L8A	25/155	PAL16L8A	25/180	
AmPAL16R8Q	35/45	PAL16R8B-4	35/55	
AmPAL16R6Q	35/45	PAL16R6B-4	35/55	
AmPAL16R4Q	35/45	PAL16R4B-4	35/55	
AmPAL16L8Q	35/45	PAL16L8B-4	35/55	
AmPAL16R8L	35/90	PAL16R8A-2	35/90	
AmPAL16R6L	35/90	PAL16R6A-2	35/90	
AmPAL16R4L	35/90	PAL16R4A-2	35/90	
AmPAL16L8L	35/80	PAL16L8A-2	35/90	
AmPAL16R8	35/180	PAL16R8A-2	35/90	Note that the MMI part requires half the power (90 mA I_{cc}) of the AMD part (180 mA)
AmPAL16R6	35/180	PAL16R6A-2	35/90	
AmPAL16R4	35/180	PAL16R4A-2	35/90	
AmPAL16L8	35/180	PAL16L8A-2	35/90	
AmPAL18P8L	35/90	PAL10H8	35/90	The 18P8L can replace any of the MMI parts, but the reverse is not necessarily true.
AmPAL18P8L	35/90	PAL12H6	35/90	
AmPAL18P8L	35/90	PAL14H4	35/90	
AmPAL18P8L	35/90	PAL16H2	35/90	
AmPAL18P8L	35/90	PAL10L8	35/90	
AmPAL18P8L	35/90	PAL12L6	35/90	
AmPAL18P8L	35/90	PAL14L4	35/90	
AmPAL18P8L	35/90	PAL16L2	35/90	

Simple Combinatorial Devices

These PAL devices provide simple combinatorial functions quickly and efficiently. . There is a wide selection of devices, allowing the user to choose the right device for the given application.

There are five basic characteristics that differentiate these devices:

- The number of inputs
- The number of outputs
- The number of product terms per output
- The speed: propagation delay
- Power consumption

Several of the devices have internal feedback, which routes output pins back as inputs. This is generally coupled with programmable three-state capability, so that the output pins can be used as outputs, inputs, or bidirectional pins.

Most of the devices have fixed output polarity. The AmPAL18P8, AmPAL22P10, PAL16P8, PAL20S10, and AmPAL22XP10 all have programmable output polarity.

The PAL20S10 has a feature called "product term steering", which allows product terms to be allocated between outputs. This is described more fully on page 5-103 of the Data Book.

The AmPAL22XP10 has built-in exclusive-OR gates on each output.

In addition to the devices that are purely combinatorial, there are also registered devices with flip-flops that can be bypassed to give a combinatorial device. Although these devices are not listed in Table 2, they can be used as combinatorial devices. The best example is the PAL22V10, which is widely used as a combinatorial device.

A summary of the architecture codes for these types of devices is shown in Table 2.

The combinatorial devices are listed in Table 3, with their logic, speed, and power characteristics.

CODE	MEANING	EXAMPLE
H	Active-HIGH outputs	PAL10H8
L	Active-LOW outputs	PAL16L8
P	Programmable output polarity	PAL16P8
C	Complementary outputs	PAL16C1
XP	Exclusive-OR gate, prog. polarity	AmPAL22XP10
S	Product term steering	PAL20S10

Table 2. Combinatorial Architectures

DEVICE NAME	INPUTS	OUTPUTS	PRODUCT TERMS/OUTPUT	SPEED (t_{PD} in ns)	STANDBY I_{CC} (mA)	DATA SHEET PAGE NO.
PAL8L14A	8	14	1	25	90	5-141
PAL6L16A	6	16	1	25	90	5-141
PAL10H8	10	8	2	35	90	5-56
PAL12H6	12	6	2,4	35	90	5-56
PAL14H4	14	4	4	35	90	5-56
PAL16H2	16	2	8	35	90	5-56
PAL10L8	10	8	2	35	90	5-56
PAL12L6	12	6	2,4	35	90	5-56
PAL14L4	14	4	4	35	90	5-56
PAL16L2	16	2	8	35	90	5-56
PAL16C1	16	2	16	40	90	5-56
PAL16L8D	16*	8	7	10	180	5-29
AmPAL16L8D				10	180	5-183
PAL16L8B				15	180	5-31
AmPAL16L8B				15	180	5-197
PALC16L8Z-25				25	0.1	5-50
PALC16L8Q-25				25	45	5-33
PAL16L8B-2				25	90	5-35
AmPAL16L8AL				25	90	5-197
PAL16L8A				25	180	5-37
AmPAL16L8A				25	155	5-197
PAL16L8B-4				35	55	5-39
AmPAL16L8Q				35	45	5-197
PAL16L8A-2				35	90	5-41
AmPAL16L8L				35	80	5-197
AmPAL16L8				35	155	5-197
PAL16L8A-4				55	50	5-43

Table 3. Simple Combinatorial PAL Devices

DEVICE NAME	INPUTS	OUTPUTS	PRODUCT TERMS/OUTPUT	SPEED (t_{PD} in ns)	STANDBY I_{CC} (mA)	DATA SHEET PAGE NO.
PAL16P8A	16*	8	7	25/30**	180	5-17
AmPAL18P8B	18*	8	8	15	180	5-202
AmPAL18P8AL				25	90	5-202
AmPAL18P8A				25	180	5-202
AmPAL18P8Q				35	55	5-202
AmPAL18P8L				35	90	5-202
PAL12L10	12	10	2	40	100	5-147
PAL14L8	14	8	2,4	40	100	5-147
PAL16L6	16	6	2,4	40	100	5-147
PAL18L4	18	4	4,6	40	100	5-147
PAL20L2	20	2	8	40	100	5-147
PAL20C1	20	2	16	40	100	5-147
PAL20L8B	20*	8	7	15	210	5-125
PAL20L8B-2				25	105	5-126
PAL20L8A				25	210	5-128
PALC20L8Z-35				35	0.1	5-133
PAL20L8A-2				35	105	5-130
PALC20L8Z-45				45	0.1	5-133
AmPAL20L10B	20*	10	3	15	210	5-306
AmPAL20L10-20				20	165	5-306
AmPAL20L10AL				25	105	5-306
PAL20L10A				30	165	5-113
PAL20S10	20*	10	0–16††	35	240	5-103
AmPAL22P10B	22*	10	8	15	210	5-306
AmPAL22P10AL				25	105	5-306
AmPAL22P10A				25	210	5-306
AmPAL22XP10-20	22*	10	2/6†	20	210	5-286
AmPAL22XP10-30L				30	105	5-286
AmPAL22XP10-30				30	180	5-286
AmPAL22XP10-40L				40	105	5-286

* Includes feedback
** Depending on polarity
† Has an exclusive-OR gate
†† Product term steering

Table 3. Simple Combinatorial PAL Devices (Continued)

Simple Registered Devices

A basic registered device is generally equivalent to a combinatorial device with registers on the outputs for signal synchronization. All such devices have feedback from the flip-flops back into the array as inputs, so that they can also be used for very fast small state machines. The differentiating characteristics of these devices are:

- The number of inputs
- The number of outputs
- The number of flip-flops
- The number of product terms per output
- The speed: maximum clock frequency
- Power consumption

Most of these devices have dedicated clock and output enable pins. On devices such as the PAL16R4, which have some dedicated combinatorial outputs, only the registered outputs are controlled by the enable pin; the combinatorial outputs have programmable three-state functions.

The PAL22V10, AmPAL22RP10 family, and AmPAL22XRP10 family have programmable three-state outputs. With these families, the clock pin may also be used as a logic input if desired.

The PAL16X4 has extra bit-pair decoding circuitry which makes it especially well suited to arithmetic operations. This is discussed more fully on page 2-53 in the combinatorial logic section.

The simpler devices all have active-LOW outputs. The PAL16RP8 family, PAL22V10, AmPAL22RP10 family, AmPAL22XRP10 family, PAL20RS10 family, and the PAL32R16 have programmable polarity.

The PAL20RS10 family and PAL32R16 have product term steering, as was discussed above for the PAL20S10.

The outputs of the PAL32R16 can also be programmed to be combinatorial in banks of eight. In addition, the PAL32R16 has multiple clock, enable, and TTL-level preload pins.

The PAL22V10 has two programmable initialization product terms: a synchronous preset term and an asynchronous reset term. These terms control all flip-flops.

A summary of the architecture codes for these types of devices is shown in Table 4. The devices are summarized in Table 5.

CODE	MEANING	EXAMPLE
R	Registered outputs	PAL16R8
X	Exclusive-OR gates	PAL16X4
RP	Registered with prog. polarity	PAL16RP8
RS	Registered with term steering	PAL20RS10
V	Versatile	AmPAL22V10

Table 4. Registered Architectures

DEVICE NAME	INPUTS	OUTPUTS	FLIP-FLOPS	PRODUCT TERMS/OUTPUT	SPEED (f_{MAX} in MHz)	STANDBY I_{CC} (mA)	DATA SHEET PAGE NO.
PAL16R8D	16*	8	8	8	55	180	5-29
AmPAL16R8D					55	180	5-183
PAL16R8B					37	180	5-31
AmPAL16R8B					40	180	5-197
PALC16R8Z-25					28.5	0.1	5-50
PALC16R8Q-25					28.5	45	5-33
PAL16R8B-2					25	90	5-35
AmPAL16R8AL					28.5	90	5-197
PAL16R8A					25	180	5-37
AmPAL16R8A					28.5	180	5-197
PAL16R8B-4					16	55	5-39
AmPAL16R8Q					18	45	5-197
PAL16R8A-2					16	90	5-41
AmPAL16R8L					18	90	5-197
AmPAL16R8					18	180	5-197
PAL16R8A-4					11	50	5-43
PAL16R6D	16*	8	6	8	55	180	5-29
AmPAL16R6D					55	180	5-183
PAL16R6B					37	180	5-31
AmPAL16R6B					40	180	5-197
PALC16R6Z-25					28.5	0.1	5-50
PALC16R6Q-25					28.5	45	5-33
PAL16R6B-2					25	90	5-35
AmPAL16R6AL					28.5	90	5-197
PAL16R6A					25	180	5-37

Table 5. Simple Registered PAL Devices

DEVICE NAME	INPUTS	OUTPUTS	FLIP-FLOPS	PRODUCT TERMS/OUTPUT	SPEED (f_{MAX} in MHz)	STANDBY I_{CC} (mA)	DATA SHEET PAGE NO.
AmPAL16R6A					28.5	180	5-197
PAL16R6B-4					16	55	5-39
AmPAL16R6Q					18	45	5-197
PAL16R6A-2					16	90	5-41
AmPAL16R6L					18	90	5-197
AmPAL16R6					18	180	5-197
PAL16R6A-4					11	50	5-43
PAL16R4D	16*	8	4	8	55	180	5-29
AmPAL16R4D					55	180	5-183
PAL16R4B					37	180	5-31
AmPAL16R4B					40	180	5-197
PALC16R4Z-25					28.5	0.1	5-50
PALC16R4Q-25					28.5	45	5-33
PAL16R4B-2					25	90	5-35
AmPAL16R4AL					28.5	90	5-197
PAL16R4A					25	180	5-37
AmPAL16R4A					28.5	180	5-197
PAL16R4B-4					16	55	5-39
AmPAL16R4Q					18	45	5-197
PAL16R4A-2					16	90	5-41
AmPAL16R4L					18	90	5-197
AmPAL16R4					18	180	5-197
PAL16R4A-4					11	50	5-43
PAL16X4	16*	8	4	8†	14	225	5-51
PAL16RP8A	16*	8	8	8	25**	180	5-17
PAL16RP6A	16*	8	6	8	25**	180	5-17
PAL16RP4A	16*	8	4	8	25**	180	5-17
PAL20R8B	20*	8	8	8	37	210	5-125
PAL20R8B-2					25	105	5-126
PAL20R8A					25	210	5-128
PALC20R8Z-35					20	0.1	5-133
PAL20R8A-2					16	105	5-130
PALC20R8Z-45					15.3	0.1	5-133
PAL20R6B	20*	8	6	8	37	210	5-125
PAL20R6B-2					25	105	5-126
PAL20R6A					25	210	5-128
PALC20R6Z-35					20	0.1	5-133
PAL20R6A-2					16	105	5-130
PALC20R6Z-45					15.3	0.1	5-133
PAL20R4B	20*	8	4	8	37	210	5-125
PAL20R4B-2					25	105	5-126
PAL20R4A					25	210	5-128
PALC20R4Z-35					20	0.1	5-133
PAL20R4A-2					16	105	5-130
PALC20R4Z-45					15.3	0.1	5-133

Table 5. Simple Registered PAL Devices (Continued)

DEVICE NAME	INPUTS	OUTPUTS	FLIP-FLOPS	PRODUCT TERMS/OUTPUT	SPEED (f_{MAX} in MHz)	STANDBY I_{CC} (mA)	DATA SHEET PAGE NO.
PAL20RS10	20*	10	10	0–16††	20	240	5-103
PAL20RS8	20*	10	8	0–16††	20	240	5-103
PAL20RS4	20*	10	4	0–16††	20	240	5-103
AmPAL22V10-15	22*	10	0–10§	8–16§§	40	180	5-249
PALC22V10H-25					33.3	90	5-79
AmPAL22V10A					28.5	180	5-260
PALC22V10H-35					20	90	5-79
AmPAL22V10					18	180	5-260
AmPAL20RP10B	22*	10	10	8	37	210	5-306
AmPAL20RP10AL					25	105	5-306
AmPAL20RP10A					25	210	5-306
AmPAL20RP8B	22*	10	8	8	37	210	5-306
AmPAL20RP8AL					25	105	5-306
AmPAL20RP8A					25	210	5-306
AmPAL20RP6B	22*	10	6	8	37	210	5-306
AmPAL20RP6AL					25	105	5-306
AmPAL20RP6A					25	210	5-306
AmPAL20RP4B	22*	10	4	8	37	210	5-306
AmPAL20RP4AL					25	105	5-306
AmPAL20RP4A					25	210	5-306
PAL32R16	32*	16	16§	0–16††	16	280	5-158

* Includes feedback
** With polarity fuse intact
† Has an exclusive-OR gate
†† Product term steering
§ Flip-flops can be bypassed
§§ Has varied product term distribution

Table 5. Simple Registered PAL Devices (Continued)

PAL Devices as Sequencers

Several registered PAL devices have added features that make them particularly well suited to state machine applications.

The basic differentiating aspects of these devices are:

- The number of inputs
- The number of outputs
- The number of flip-flops
- The type of flip-flop possible (D, T, J-K, S-R, latch, and/or buried)
- The number of product terms per output
- The speed: maximum clock frequency
- Power consumption

The PAL20X10 series, AmPAL20XRP10 series, PAL22RX8, and PAL32VX10 all have an exclusive-OR gate that makes it possible to design with D, T, J-K, or S-R flip-flops. Since the PAL20X10 has fewer product terms, it is best for counter-like applications. The AmPAL20XRP10 series, PAL22RX8 and PAL32VX10 have more product terms, for much more sophisticated state machines.

The AmPAL23S8 has six extra flip-flops that are permanently buried in addition to eight output flip-flops. The PAL32VX10 and AmPALC29M16 also have flip-flops that can be buried if they are not needed at the outputs. An output pin can be used as an input if its flip-flop is buried. All of these devices are ideal for applications incorporating internal state machines as part of the overall design.

The PAL22RX8, PAL32VX10, and AmPAL23S8 have global preset and reset control, making system initialization much easier.

The AmPALC29M16 is an electrically erasable device that has an unusually flexible architecture. Almost all pins can be used as inputs or outputs; both inputs and outputs can be registered, latched, or combinatorial. There are two pins that can be used as clocks. Global preset and reset functions are also provided. Several of the flip-flops can be buried to provide internal state machines. This device is HC and HCT compatible.

A summary of the architecture codes for these types of devices is shown in Table 6.

The PAL devices best suited to state machine applications are listed in Table 7.

CODE	MEANING	EXAMPLE
X	Exclusive-OR gates	PAL20X10
XRP	Exclusive-OR gates, prog. polarity	AmPAL20XRP10
RX	Registered, exclusive-OR gates	PAL22RX8
S	Sequencer	AmPAL23S8
M	Advanced macrocell	AmPALC29M16
VX	Varied term distribution, XOR gate	PAL32VX10

Table 6. State Machine Architectures

DEVICE NAME	INPUTS	OUTPUTS	FLIP-FLOPS	FLIP-FLOP TYPES	PRODUCT TERMS/OUTPUT	SPEED (f_{MAX} in MHz)	STAND BY I_{CC} (mA)	DATASHEET PAGE NO.
PAL20X10A	20*	10	10	D,T,JK,SR	2/2†	22.2	180	5-113
PAL20X8A	20*	10	8	D,T,JK,SR	2/2†	22.2	180	5-113
PAL20X4A	20*	10	4	D,T,JK,SR	2/2†	22.2	180	5-113
AmPAL20XRP10-20	22*	10	10	D,T,JK,SR	2/6†	30.3	210	5-286
AmPAL20XRP10-30L						22.2	105	5-286
AmPAL20XRP10-30						22.2	180	5-286
AmPAL20XRP10-40L						14.3	105	5-286
AmPAL20XRP8-20	22*	10	8	D,T,JK,SR	2/6, 8†	30.3	210	5-286
AmPAL20XRP8-30L						22.2	105	5-286
AmPAL20XRP8-30						22.2	180	5-286
AmPAL20XRP8-40L						14.3	105	5-286
AmPAL20XRP6-20	22*	10	6	D,T,JK,SR	2/6, 8†	30.3	210	5-286
AmPAL20XRP6-30L						22.2	105	5-286
AmPAL20XRP6-30						22.2	180	5-286
AmPAL20XRP6-40L						14.3	105	5-286
AmPAL20XRP4-20	22*	10	4	D,T,JK,SR	2/6, 8†	30.3	210	5-286
AmPAL20XRP4-30L						22.2	105	5-286
AmPAL20XRP4-30						22.2	180	5-286
AmPAL20XRP4-40L						14.3	105	5-286
PAL22RX8A	22*	8	8§	D,T,JK,SR	1/8†	28.5	210	5-87
AmPAL23S8-20	23*	8	14§	D,B◊	6–12§§	33	200	5-169
AmPAL23S8-25						28.5	200	5-169
AmPALC29M16-35	29*	16	16§	D,B,L◊	8–16§§	20	120	5-231
AmPALC29M16-45						15	120	5-231
PAL32VX10A	32*	10	10§	D,T,JK,SR, B◊	1/8–16†	25	180	5-70
PAL32VX10						22.2	180	5-70

* Includes feedback
† Has an exclusive-OR gate
§ Some flip-flops can be bypassed
§§ Has varied product term distribution
◊ B=flip-flops are or can be buried; L=latched outputs possible

Table 7. State Machine PAL Devices

Asynchronous PAL Devices

Several PAL devices have been designed for applications that require asynchronous control of flip-flops. The simplest devices are the PAL20RA10 and the PAL16RA8. These devices essentially consist of several flip-flops driven by a programmable AND array and a fixed OR array. Thus the D-type flip-flops have individually programmable clock, set, and reset lines. They also have programmable polarity, and the flip-flops can be bypassed to give combinatorial outputs.

These devices are also useful for replacing random discrete logic. Each flip-flop is independent; the clocks can be gated if desired, as can the set and reset lines. This provides for flexibility both when replacing random gates and when designing logic that is asynchronous in nature.

The AmPALC29MA16 combines some of the advantages of the AmPALC29M16 with the advantages of the PAL20RA10. Most of the AmPALC29M16 macrocell is preserved, except that each flip-flop is given a programmable clock, set, and reset line. Each output also has individual three-state control. This device is electrically erasable, and is HC and HCT compatible.

A summary of the architecture codes for these types of devices is shown in Table 8. The asynchronous devices available are summarized in Table 9.

CODE	MEANING	EXAMPLE
RA	Registered, asynchronous	PAL20RA10
MA	Asynchronous macrocell	AmPALC29MA16

Table 8. Asynchronous Architectures

CMOS PAL Devices

PAL devices have recently become available in CMOS technology, providing performance competitive with that of many bipolar devices, but with reduced power consumption. The functions of these devices have been discussed above, along with the TTL devices. The particular characteristics of the CMOS versions are discussed below.

Zero-Power vs Low-Power

Two basic types of CMOS PAL devices are available: those that dissipate essentially no power when in a quiescent state, and faster devices which draw a nominal amount of current even when quiescent. Devices are thus classified as "zero-power" or "low-power". The low-power devices still require far less current than their bipolar equivalents.

The basic difference between zero-power and low-power devices is the wake-up circuitry used in the zero-power devices. Any time a signal changes, the device wakes up in order to respond to the signal. After the transition is complete, the device goes "back to sleep", as long as no other signals are changing. This wake-up circuitry exacts a slight speed penalty, usually about 5 ns, but in applications where minimal power dissipation is crucial, the benefits of zero-power operation greatly outweigh the extra delay.

The PALC16R8Z family actually has a "turbo product term". This product term allows the device to operate either in zero-power or low-power mode. Since this choice has been provided by a product term, it is actually possible to have a device idle in zero-power mode, and then switch into low-power mode for faster operation. This is particularly appropriate for devices that tend to operate in bursts, with idle periods between bursts.

HC vs HCT

CMOS devices can be characterized as being HC and/or HCT compatible. HC-compatible devices have outputs that swing "rail-to-rail", or from ground to VCC. They also have an input threshold which is about 2.5 V. These devices are normally used when driving other CMOS devices.

DEVICE NAME	INPUTS	OUTPUTS	PRODUCT TERMS/OUTPUT	SPEED (t_{PD} in ns)	STANDBY I_{CC} (mA)	DATA SHEET PAGE NO.
PAL16RA8	16*	8	4	30**	170	5-11
PAL20RA10-20	20*	10	4	20**	200	5-95
PAL20RA10				30**	200	5-97
AmPALC29MA16-35	29*	16	4–12††	35	120	5-209
AmPALC29MA16-45				45	120	5-209

* Includes feedback
** With polarity fuse intact
†† Has product term steering

Table 9. Asynchronous PAL Devices

HCT-compatible devices have TTL-level output voltages; that is, a VOH of about 3.5 V and a VOL of about 0.5 V. The input threshold is usually adjusted to 1.5 V, which is the threshold of TTL devices. These devices are more suitable for a mixed TTL and CMOS system.

Roughly speaking, HC and HCT devices can be interchanged. However, since HCT outputs do not swing rail-to-rail, if they drive another CMOS device, that device will draw more power. HCT outputs are faster, have more drive, and are more latch-up immune. The input threshold is less critical, and is important only in very touchy situations.

Note that some devices are both HC and HCT compatible. This means that they have a quick, high drive output rise to a TTL VOH, and then a slower rise to VCC, ultimately providing rail-to-rail levels. This is illustrated in Table 10.

FAMILY	INPUT THRESHOLD (V)	V_{OL} (V)	V_{OH} (V)
HC	2.5	0	Vcc
HCT	1.5	0.5	3.5
HC/HCT	1.5	0	3.5 -> Vcc

Table 10. CMOS/TTL Compatibility

UV Erasable vs Electrically Erasable

CMOS PAL devices come in one of two basic technologies:

- UV erasable
- Electrically erasable

UV-erasable devices use EPROM technology in implementing the array. They can be erased by exposure to ultraviolet light if the device is packaged in a windowed package. These packages are more expensive, so the devices are also available in lower-cost plastic packages. UV-erasable devices in plastic packages are not erasable, and are thus referred to as "One-Time Programmable", or OTP devices.

Electrically erasable devices can be erased by providing the appropriate signals to the device. No window is required in the package, and the erase time is much faster than the erase time of UV-erasable devices. Erasure is implemented by the device programmer as a portion of the overall programming algorithm.

Summary of CMOS PAL Devices

The PAL devices available in CMOS have been described along with their bipolar counterparts in the preceding pages. They are summarized in Table 11, along with their power, compatibility, and erasability characteristics.

DEVICE NAME	INPUTS	OUTPUTS	SPEED (t_{PD} in ns)	STANDBY I_{CC} (mA)	COMPATIBILITY	ERASABILITY	DATA SHEET PAGE NO.
PALC16L8Z-25	16*	8	25	0.1/45**	HC/HCT	UV	5-50
PALC16R8Z-25	16*	8	25	0.1/45**	HC/HCT	UV	5-50
PALC16R6Z-25	16*	8	25	0.1/45**	HC/HCT	UV	5-50
PALC16R4Z-25	16*	8	25	0.1/45**	HC/HCT	UV	5-50
PALC16L8Q-25	16*	8	25	45	HCT	UV	5-33
PALC16R8Q-25	16*	8	25	45	HCT	UV	5-33
PALC16R6Q-25	16*	8	25	45	HCT	UV	5-33
PALC16R4Q-25	16*	8	25	45	HCT	UV	5-33
PALC20L8Z-35	20*	8	35	0.1	HC/HCT	UV	5-133
PALC20L8Z-45	20*	8	45	0.1	HC/HCT	UV	5-133
PALC20R8Z-35	20*	8	35	0.1	HC/HCT	UV	5-133
PALC20R8Z-45	20*	8	45	0.1	HC/HCT	UV	5-133
PALC20R6Z-35	20*	8	35	0.1	HC/HCT	UV	5-133
PALC20R6Z-45	20*	8	45	0.1	HC/HCT	UV	5-133
PALC20R4Z-35	20*	8	35	0.1	HC/HCT	UV	5-133
PALC20R4Z-45	20*	8	45	0.1	HC/HCT	UV	5-133
PALC22V10H-25	22*	10	25	90	HCT	UV	5-79
PALC22V10H-35	22*	10	35	90	HCT	UV	5-79
AmPALC29M16-35	29*	16	35	120	HC/HCT	EE	5-231
AmPALC29M16-45	29*	16	45	120	HC/HCT	EE	5-231
AmPALC29MA16-35	29*	16	35	120	HC/HCT	EE	5-209
AmPALC29MA16-45	29*	16	45	120	HC/HCT	EE	5-209

* Includes feedback
** With turbo term

Table 11. Summary of CMOS PAL Devices

ECL PAL Devices

For a long period of time, PLDs were available only for TTL-compatible systems. More recently, however, PAL devices have been introduced which address the users of ECL logic.

Two basic families of ECL logic are addressed by the ECL PAL devices. These families differ in their voltage ranges and the compensation built into the circuitry (Table 12). These differences also affect noise margins and thresholds. Refer to page 3-150 for more information on ECL design.

The ECL PAL devices presently have a 6-ns propagation delay; the differences between the devices can be found in the architectures. The PAL10H20P8 and PAL10H20G8 are simpler devices which are compatible with other 10KH logic; the former is a combinatorial device with programmable polarity, the latter has output latches. Both devices have product term steering for added flexibility in allocating logic resources. These features are described in more detail on page 5-382.

The PAL10H20EV/EG8 and PAL10020EV/EG8 are 10KH and 100K compatible options of the same device, respectively. These devices have more product terms than the first two devices, and have outputs that can be programmed as combinatorial or registered/latched. These devices also have global reset and preset controls.

A summary of the architecture codes for these types of devices is shown in Table 13. The ECL PAL devices and their basic characteristics are summarized in Tables 14 and 15.

FAMILY	V_{EE}	COMPENSATION
10KH	-5.2V ± 5%	Voltage
100K	-4.5V ± 0.3V	Voltage, temperature

Table 12. 10KH ECL vs 100K ECL

Programmable Sequencers

Many PAL devices can be used to design very fast, efficient sequencers. Optimal among these devices are the PAL32VX10 and the AmPAL23S8. However, there are additional devices with entire architectures dedicated to the task of sequencing.

Since these devices are primarily intended for use as state machines, the important characteristics to be evaluated are:

- The number of inputs
- The number of outputs
- The number of states that can be implemented
- The number of branches available from any state
- The speed: maximum clock frequency
- Power consumption

The programmable sequencers available are shown in Table 16.

The PMS14R21 (also known as the PROSE™ device) actually consists of a PAL device for branching, followed by a PROM to store the states. This device also has the Diagnostics-On-Chip™ (DOC™) testability circuitry, allowing the device to be inserted into a serial scan path. This is described more thoroughly on page 3-123 of the testability section.

The PLS105, PLS167, and PLS168 are based on the standard PLS architecture, where both the AND array and the OR array are programmable. S-R flip-flops are used to minimize the need for specifying 'HOLD' conditions. There is also a "complement array", which makes it easier to define default state transitions.

CODE	MEANING	EXAMPLE
P	Combinatorial, prog. polarity	PAL10H20P8
G	Latched outputs	PAL10H20G8
EG	Latched or combinatorial	PAL10H/10020EG8
EV	Registered or combinatorial	PAL10H/10020EV8

Table 13. ECL PAL Device Architectures

DEVICE NAME	INPUTS	OUTPUTS	FLIP-FLOPS	PRODUCT TERMS/OUTPUT	SPEED (t_{PD} or f_{MAX})	I_{EE} (mA)	DATA SHEET PAGE NO.
PAL10H20P8	20*	8	0	0–8††	6 ns	210	5-385
PAL10H20G8	20*	8	8§	0–8††	6 ns	225	5-382
PAL10H20EV/EG8	20*	8	8§	8–12§§	125 MHz	220	5-381

* Includes feedback
§ Flip-flops can be bypassed
†† Has product term steering
§§ Has varied product term distribution

Table 14. 10KH-Compatible PAL Devices

DEVICE NAME	INPUTS	OUTPUTS	FLIP-FLOPS	PRODUCT TERMS/OUTPUT	SPEED (f_{MAX})	I_{EE} (mA)	DATA SHEET PAGE NO.
PAL10020EV/EG8	20*	8	8§	8–12§§	125 MHz	220	5-381

* Includes feedback
§ Flip-flops can be bypassed
§§ Has varied product term distribution

Table 15. 100K-Compatible PAL Devices

The PMS14R21, PLS105, PLS167, and PLS168 are all supported by PALASM software, using a convenient entry format that has been optimized for state machines. This essentially makes it possible to transfer a state diagram directly to a design file without the need for manual logic transformations or calculations.

The Am29PL141 Fuse Programmable Controller (FPC) is an instruction-based sequencer. This is especially convenient for designers who are more comfortable using instruction-based devices. The design is implemented more as microprocessor program might be written. Indeed, the architecture itself, which is described in more detail on page 5-339, resembles that of a simple processor. This device has its own assembler which is used to reduce the sequencer "program" into a file that can be used to program the device.

The Am2971 is a Progammable Event Generator. It is a versatile timing device that can be used as a digital substitute for analog delay lines or as a general-purpose user-programmable timing/waveform generator.

LCA Devices

Large-scale designs are now possible with the introduction of the Logic Cell Array (LCA) devices. These devices can be thought of as programmable gate arrays, and provide an efficient, cost-effective means of implementing dense designs within a single PLD.

Unlike other PLDs, which are non-volatile, LCA devices make use of static RAM technology. This makes them easy to develop, easy to test and debug, and reprogrammable.

There is an entire family of support software and hardware packages available to aid in the design of LCA applications. These are described in more detail on page 3-17.

The architecture of an LCA device resembles that of a gate array more closely than a traditional PLD. It has an array of Configurable Logic Blocks (CLBs) that can be connected via a variety of interconnect resources. There are also Input/Output Blocks (IOBs) for transferring the signals on and off the chip.

The main features that distinguish one LCA device from another are:

- The number of I/O pins
- The number of Configurable Logic Blocks
- The speed: internal maximum toggle frequency
- Power consumption

The available devices are summarized in Table 17.

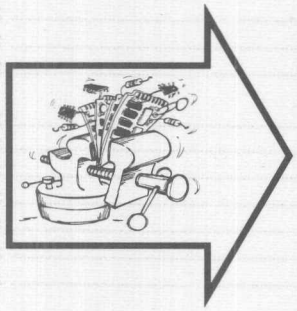
DEVICE NAME	INPUTS	OUTPUTS	STATES (MAX)	BRANCHES PER STATE	SPEED (f_{MAX} in MHz)	I_{CC} (mA)	DATA SHEET PAGE NO.
PMS14R21A	8	8	128	4	30	210	5-315
PMS14R21					25	210	5-315
PLS105-37	16	8	<64*	*	37	200	5-331
PLS167-33	14	6	<128*	*	33	200	5-331
PLS168-33	12	8	<1028*	*	33	200	5-331
Am29PL141	6	16	64	2	20	450	5-339
Am2971	6	14	N/A	N/A	85	310	5-365

* Depends highly on state diagram topology. May be limited by number of product terms or number of flip-flops.

Table 16. Programmable Sequencers

DEVICE NAME	I/O PINS	CLBs	SPEED (INTERNAL TOGGLE f_{MAX} in MHz)	STANDBY I_{CC} (mA)	DATA SHEET PAGE NO.
M2064-70	58	64	70	5	5-483
M2064-50			50	5	5-483
M2064-33			33	5	5-483
M2018-50	74	100	50	5	5-483
M2018-33			33	5	5-483

Table 17. LCA Devices



PAL Device Handbook

Introduction 1

Applications 2

PAL Device Data Book

Programming and Quality 3

PALASM 2 Software User Documentation 4

Data Sheets 5

Appendices 6

Beginner's Guide 2-1
Constructing a Combinatorial Design 2-2
Constructing a Registered Design 2-9
Programming a Device 2-14

PLD Design Methodology 2-21
Conceptualizing a Design 2-22
Device Selection Considerations 2-23
Implementing a Design 2-26
Simulation 2-30
Device Programming and Testing 2-33

Combinatorial Logic Design 2-35
Encoders and Decoders 2-35
Multiplexers 2-43
Comparators 2-45
Range Decoders 2-52
Adders/Arithmetic Circuits 2-53
Latches 2-58

Registered Logic Design 2-61
Binary Counters 2-67
Modulo Counters 2-75
Gray-Code Counters 2-87
Johnson Counters 2-88
Shift Registers 2-90
Asynchronous Registered Designs 2-94

State Machine Design 2-101
State Machine Theory 2-103
State Machine Types: Mealy & Moore 2-105
Device Selection Considerations 2-107
PAL Devices as Sequencers 2-111
Programmable Logic Sequencers (PLS) 2-117
PROSE Sequencer (PMS14R21) 2-120
Fuse Programmable Controller (Am29PL141) 2-121
State Machine Design Tutorial 2-122

Microprocessor-Based Systems 2-131
Interfacing to the 8086/80186/80286 2-134
8086 and Am7990 LANCE Interface 2-135
8086 and Am9516 Universal DMA Controller Interface 2-138
80286 to Am9568 Data Ciphering Processor Interface 2-143
80286 to Am8530 Interface 2-147
Interfacing to the 68000/68020 2-149
The 68000 and Am8530 Interface with Interrupts 2-150
68000 and Am7990 LANCE Interface 2-153
68000 to AmZ8068 Data Ciphering Processor Interface 2-155
68000 and Dual Am9516 DMA Controllers Interface 2-159
Am8530 to 68020 Interface 2-162
Interfacing to the 8088 2-165
8088 to Am9516 UDC Interface 2-166
80186 to Am9516 Universal DMA Controller Interface 2-170
68000 Interrupt Controller 2-172

Memory Control 2-179
Memory Handshake Logic 2-184
Customize a DRAM Controller Using Advanced PAL Devices 2-187
8088 to Am2968 Interface 2-202
MC68000 to Am2968 Interface 2-210
General-Purpose Dual-Port Arbiter 2-215
Dynamic Memory Control State Sequencer 2-224
8-Bit Error Detection and Correction 2-229
Fuse Programmable Controller Simplifies Cache Design 2-239
PAL22RX8A Provides Control and Addressing for a 32-Location-Deep RAM-Based LIFO 2-250

Graphics and Image Processing Systems 2-257
Small System Video Controller 2-261
PAL32VX10 Uses Buried Register for Input-Intensive State Machine Designs 2-275

Digital Signal Processing 2-283
Waveform Generator 2-286
PAL32VX10 Uses Buried Register for Input-Intensive State Machine Designs 2-292
Analog to Digital Conversion 2-297
PAL Devices, PROMs, FIFOs and Multipliers Team up to
Implement Single-Board High-Performance Audio Spectrum Analyzer 2-307

Bus Interface 2-325
Unibus Interrupt Controller 2-328
A MULTIBUS Arbiter Design for 10 MHz Processors 2-333
MULTIBUS to Am9516 Interface 2-338
Z-BUS and 8088/8086 Interface 2-342
VME Bus Control Simplified with PLDs 2-347

2

Communications 2-357
- B8ZS Coding Using CMOS ZPAL Devices 2-362
- HDB3 Line Coding Using PAL Devices 2-384
- ZPAL Devices Implement D4 Frame Synchronization 2-404
- T1 Extended Superframe Provides Transmission Error Detection 2-431
- Time Division Multiplexing with the LCA Device 2-435
- LCA Device Implements an 8-Bit Format Converter in a PBX Switching Module 2-444
- Serial Data Link Controller 2-453
- QAM Encoder in a ZPAL Device 2-456
- PAL Devices Implement the Full V.32 Convolution Encoder 2-463
- PLD Devices Implement 4B3T Line Transcoder 2-475
- PALC22V10 Creates Manchester Encoder Circuit 2-499

Peripheral Design 2-507
- Building an ESDI Translator Using the M2064 Logic Cell Array 2-509
- Writing a Tape Drive Controller in PROSE 2-519
- GCR (4B-5B) Encoder/Decoder 2-541

Industrial Control 2-547
- Stepper Motor Controllers 2-550
- Shaft Encoders 2-558

Military Applications 2-579
- Radiation Hardness 2-581

Article Reprints
- Programmable Logic Device Preserves Pins, Product Terms (PAL32VX10) 2-589
- PLD Programmability Extends its Sway Over Complex I/O (PALC29M/MA16) 2-593
- PROSE Devices Simplify State Machine Design (PMS14R21) 2-601
- Designing a State Machine with a Programmable Sequencer (PMS14R21) 2-607
- FPCs and PLDs Simplify VME Bus Control (Am29PL141) 2-619
- Fuse-Programmable Chip Takes Command of Distributed Systems (Am29PL141) 2-633
- PAL Device Buries Registers, Brings State Machines to Life (AmPAL23S8) 2-639
- Programmable Event Generator Conquers Timing Restraints (Am2971) 2-644
- Wait-State Remover Improves System Performance (PAL22V10) 2-648
- PLDs Implement Encoder/Decoder for Disk Drives (PAL22V10) 2-651
- Mixing Data Paths Expands Options in System Design (PAL22V10) 2-660
- Programmable Logic Chip Rivals Gate Arrays in Flexibility (PAL22V10) 2-670
- XOR PLDs Simplify Design of Counters and Other Devices (PAL20X10) 2-676
- The PAL20RA10 Story —The Customization of a Standard Product (PAL20RA10) 2-683
- PLDs Abound: RAM-Based Logic Joins In 2-699
- Introduction to Programmable Array Logic 2-702
- Logical Alternatives in Supermini Design 2-711

Conference Proceedings
- New PAL Device Architecture Extends Design Flexibility (PAL32VX10) 2-719
- PROSE Architecture and Design Methodology (PMS14R21) 2-724
- Blazing Fast PAL Devices Enable New Application Areas (PAL16R8-10, PAL10H20P/G8) 2-730

Sales Offices 2-740

PLD Applications by Product

All design files use PALASM 2 software unless otherwise noted.

PAL23S8 (PLPL Software)
Customize a DRAM Controller (2) 2-187, 2-639

PAL16RA8
HDB3 Line Coding Using PAL Devices 2-384

PAL16P8 (See PAL18P8)
PAL16RP8 (See PAL16RP6)
PAL16RP6
Memory Handshake Logic 2-184
PAL16RP4
4-Bit Counter 2-69

PAL16L8
Address Decoder 2-22
8086 and Am7990 LANCE Interface 2-135
8086 and Am9516 Universal DMA Controller Interface 2-138
80186 to Am9516 Universal DMA Controller Interface 2-170
68000 and Am7990 LANCE Interface 2-153
MC68000 to Am2968 Interface 2-210
General-Purpose Dual-Port Arbiter (2) 2-215
8-Bit Error Detection and Correction (2) 2-229
Audio Spectrum Analyzer 2-307
MULTIBUS to Am9516 Interface 2-338
ZPAL Devices Implement D4 Frame Synchronization 2-404

PAL16R8
Basic Flip-Flops 2-17
Dual BCD Counter 2-77
Dynamic Memory Control State Sequencer 2-224
Z-Bus and 8088/8086 Interface 2-342
B8ZS Coding Using CMOS ZPAL Devices 2-362
Stepper Motor Controllers 2-550
Shaft Encoders 2-558

PAL16R6
8088 to Am2968 Interface 2-202
Fuse Programmable Controller Simplifies Cache Design 2-239
Audio Spectrum Analyzer (2) 2-307
B8ZS Coding Using CMOS ZPAL Devices (2) 2-362
ZPAL Devices Implement D4 Frame Synchronization 2-404
T1 Extended Superframe Provides Transmission Error Detection 2-431
PLD Devices Implement 4B3T Line Transcoder (2) 2-475
GCR (4B-5B) Encoder/Decoder 2-541

PAL16R4
8086 and Am7990 LANCE Interface 2-135
8086 and Am9516 Universal DMA Controller Interface 2-139
80286 to Am9568 Data Ciphering Processor Interface 2-143
80286 to Am8530 Interface (PLPL) 2-147
The 68000 and Am8530 Interface with Interrupts 2-150
68000 to AmZ8068 Data Ciphering Processor Interface 2-155
Am8530 to 68020 Interface (PLPL) 2-162
68000 Interrupt Controller 2-172
8088 to Am2968 Interface 2-202
A MULTIBUS Arbiter Design for 10 MHz Processors 2-333
MULTIBUS to Am9516 Interface 2-338
HDB3 Line Coding Using PAL Devices 2-384
Stepper Motor Controllers 2-550
Shaft Encoders 2-558

PAL16X4 (PALASM 1 Software)
Between Limits Comparator/Register 2-47
Adder 2-57
8-Bit Error Detection and Correction (2) 2-229

PAL18P8
General-Purpose Dual-Port Arbiter 2-215
Fuse Programmable Controller Simplifies Cache Design 2-239
PLDs Implement Encoder/Decoder for Disk Drives (16HD8) 2-651

PAL10H8 (See PAL18P8)
PAL12H6
Basic Gates 2-15
Audio Spectrum Analyzer (2) 2-307
PAL14H4 (See PAL18P8)
PAL16H2
Simple Encoder 2-36
PAL16C1
Octal Comparator 2-50
Audio Spectrum Analyzer 2-307
PAL10L8
Audio Spectrum Analyzer (2) 2-307
PAL12L6 (See PAL18P8)
PAL14L4
Quad 3:1 Multiplexer 2-44
PAL16L2 (See PAL18P8)

PALC29MA16 (PLPL Software)
PLD Programmability Extends its Sway Over Complex I/O 2-593
PALC29M16 (PLPL Software)
PLD Programmability Extends its Sway Over Complex I/O 2-593

PAL32VX10
Small System Video Controller (3) 2-261
Dual Port Video Shift Register 2-275
Dual Binary Rate Multipliers 2-292
Programmable Logic Device Preserves Pins, Product Terms 2-589
New PAL Device Architecture Extends Design Flexibility 2-719

PAL22V10
Modulo-360 Counter (PLPL) 2-81
Nine-Bit Johnson Counter (PLPL) 2-89
Eight-Bit Barrel Shifter (PLPL) 2-93
68000 and Dual Am9516 DMA Controllers Interface 2-159
8088 to Am9516 UDC Interface (PLPL) 2-166
General-Purpose Dual-Port Arbiter (PLPL) (3) 2-215
Audio Spectrum Analyzer 2-307
VME Bus Control Simplified with PLDs (PLPL) (2) 2-347, 2-619
PLD Devices Implement 4B3T Line Transcoder 2-475
PALC22V10 Creates Manchester Encoder Circuit 2-499
Wait-State Remover Improves System Performance 2-648
PLDs Implement Encoder/Decoder for Disk Drives 2-651
Mixing Data Paths Expands Options in System Design 2-660
Programmable Logic Chip Rivals Gate Arrays in Flexibility 2-670

PAL22RX8
PAL22RX8A Provides Control and Addressing for a 32-Location Deep RAM-Based LIFO 2-250

PAL20RA10
5-Bit Ripple Counter 2-94
Frequency Divider 2-99
Unibus Interrupt Controller 2-328
Serial Data Link Controller 2-453
The PAL20RA10 Story: The Customization of a Standard Product 2-683

PAL22XP10 (See PAL20L10)
PAL20XRP10 (See PAL20X10)
PAL20XRP8 (See PAL20X8)
PAL20XRP6 (See PAL20X8)
PAL20XRP4 (See PAL20X8)

PAL20S10 (See PAL20L10)
PAL20RS10
Analog to Digital Conversion 2-297
PAL20RS8
HDB3 Line Coding Using PAL Devices 2-384
PAL Devices Implement the Full V.32 Convolution Encoder 2-463
PAL20RS4 (See PAL20RS8)
HDB3 Line Coding Using PAL Devices 2-384
PAL20L10
Clean Octal Latch 2-59
68000 Interrupt Controller 2-172
Unibus Interrupt Controller 2-328

PAL20X10
9-Bit Counter 2-72
Waveform Generator 2-286
Shaft Encoders 2-558
XOR PLDs Simplify Design of Counters and Other Devices 2-676

PAL20X8
Universal Shift Register 2-91
PAL Devices, PROMs, FIFOs, and Multipliers Team up to Implement Single-Board High-Performance Audio Spectrum Analyzer 2-307
PAL Devices Implement the Full V.32 Convolution Encoder 2-463
XOR PLDs Simplify Design of Counters and Other Devices 2-676

PAL22P10 (See PAL18P8)
PAL20RP10 (See PAL20RS10)

PAL20RP8 (See PAL20RS8)
PAL20RP6 (See PAL20RS8)
PAL20RP4 (See PAL20RS8)

PAL20L8
Writing a Tape Drive Controller in PROSE 2-519
PAL20R8
QAM Encoder in a ZPAL Device 2-456
PAL20R6 (See PAL20R8)
PAL20R4
16-Input Registered Priority Encoder 2-37
ZPAL Devices Implement D4 Frame Synchronization 2-404

PAL6L16
Four-to-Sixteen Decoder 2-40
PAL8L14 (See PAL6L16)

PAL12L10 (See PAL18P8)
PAL14L8 (See PAL18P8)
PAL16L6 (See PAL18P8)
PAL18L4 (See PAL18P8)
PAL20L2 (See PAL18P8)
PAL20C1 (See PAL18P8)

PAL32R16
Addressable Register 2-100
Analog to Digital Conversion 2-297

PMS14R21
Traffic Signal Controller 2-128
Writing a Tape Drive Controller in PROSE 2-519
PROSE Devices Simplify State Machine Design 2-601
Designing a State Machine with a Programmable Sequencer 2-607
PROSE Architecture and Design Methodology 2-724

PLS167 (See PMS14R21 and 2-117)
PLS168 (See PMS14R21 and 2-117)
PLS105 (See PMS14R21 and 2-117)

2971 (PEGASUS Software)
Programmable Event Generator Conquers Timing Constraints 2-644

29PL141 (ASM14X Software)
Fuse Programmable Controller Simplifies Cache Design 2-239
FPCs and PLDs Simplify VME Bus Control 2-347, 2-619
Fuse-Programmable Chip Takes Command of Distributed Systems 2-633

PAL10020EV/EG8 (See PAL22V10)
PAL10H20EV/EG8 (See PAL22V10)
PAL10H20G8 (See PAL16R8)
PAL10H20P8 (See PAL18P8)

M2064 (XACT Software)
Time Division Multiplexing with the LCA Device 2-435
Building an ESDI Translator Using the M2064 Logic Cell Array 2-509

M2018 (XACT Software)
LCA Device Implements an 8-Bit Format Converter in a PBX Switching Module 2-444

Beginner's Guide

Introduction

This section is intended as a beginner's introduction to PLD design, although experienced users may find it a good review. We will take a step-by-step approach through two very simple designs to demonstrate the basic PLD design implementation process.

By "beginner", we mean a logic designer who is just beginning to use programmable logic. You may have a lot of experience with discrete digital logic, or you may have just graduated from college. We assume a basic understanding of digital logic, although there is a logic reference on page 6-1 should you need a refresher. Some computer experience is helpful, but not essential.

Through this effort, you will be introduced to PALASM software and to the concept of device programming. Because of the simplicity of the designs, we will be using PAL devices, but the implementation concepts will be applicable to all of Monolithic Memories' and Advanced Micro Devices' PLDs.

Only enough detail of the software will be presented to allow you to implement the examples. This will also give you enough information to understand most of the other design examples in this handbook. More software details can be found in the actual software documentation in section 4.

To keep the discussion simple, we will make some assumptions about the system being used to process your designs. We will assume an IBM PC/XT™ or compatible system with one hard disk and one floppy drive, running PC-DOS™ or MS-DOS™ version 2.1 or higher. The installation procedure for the software will be described for this setup. The procedures for other setups and machines are not radically different; details can be found in the PALASM software documentation, section 4.

We will also take no significant shortcuts for these examples, even though there may be times when we could. In this way, you can gain a better understanding of exactly what is happening as you implement your design.

We will talk about device programming, describing all of the steps that are necessary to program a PLD. However, due to the wide variety of programmers available, we will not get down to the level of detail that tells you exactly which buttons to push. Although we will get as close as we can, we must defer the details to your programmer manual.

As we work the design examples, we will, of course, have to choose devices into which to put the designs. However, we will not dwell here on the parameters that go into choosing a device. This is covered more thoroughly in the rest of this section.

Installing PALASM Software

Before we actually start implementing a design, we must install PALASM software. If you already have the software installed, you may disregard this discussion. The procedure described here will allow you to install the software package onto an IBM PC/XT or compatible with a hard disk and at least one floppy drive.

After you have turned on the computer and allowed it to boot up, you can install the software by placing disk #1 into drive A, and typing:

```
A:PAL2INST<CR>
```

If you are new to the PC, <CR> is the 'RETURN' key on the right side of the keyboard.

The program will ask you which disk you wish to install the programs on. Enter the disk name, or just hit <CR> if installing onto disk C. The installation program creates directories for all of the programs. If you presently have an older version of PALASM software installed in a similar directory structure, the installation program will warn you that the contents of the directories will be changed. Otherwise, it will start installing.

Note that the installation procedure may modify two files which are used when the computer boots up: AUTOEXEC.BAT and CONFIG.SYS. These changes should not affect any other programs you run.

The installation program will then give you a choice of programs to install; hit

```
1<CR>
```

to install the software. The installation program will ask you to insert various disks as it installs the programs. The disks are labelled, so this should be a simple procedure. After the software has been installed, reboot the system by holding down

```
<CTRL><ALT><DEL>
```

at the same time. This only needs to be done after installation. In the future when you turn the computer on, the computer will boot automatically.

You can now call up the menu by typing

```
PALASM<CR>
```

Now you need to install the editor or word processing program you wish to use. You may use any editor you wish, as long as it can generate a clean ASCII text file. In addition, you need to

install a communication program; any one will do. A simple one called PC2 is provided with the PALASM software. You can install these programs by hitting

```
<F6>
```

When the setup screen comes up, hit the down-arrow key until it gets to the line that says "Editor". Enter the directory path and program name for the editor you wish to use. For example, if you use WordStar™, and you have the program located in the directory called "WS" on disk C, then enter

```
C:\ws\ws
```

Add enough spaces to delete whatever was there before.

This takes you into the next field. Here you need to enter the filetype of your editor, which will be either .COM or .EXE.

Once this is complete, for the WordStar example, the line should now look like:

```
C:\ws\ws        .com
```

Hit <CR> again to get to the next field. Here you enter the name of your communication software. If you wish to use PC2, then type

```
\PALASM\SUPL\PC2
(adding spaces to delete what was there before.)
<CR>
.EXE
```

If you are not using PC2, then replace "PC2" with the name of your program. This procedure works regardless of the editor or communication software you wish to use. When you are finished, hit

```
<ESC>
```

to get back to the menu.

You are now ready to start.

Constructing a Combinatorial Design —Basic Gates

The first example we will try is a very simple combinatorial circuit consisting of all of the basic logic gates, as shown in Figure 1. This will be helpful for those designs where you are integrating random logic into a PAL device to save space and money.

As can be seen from the figure, there will be six separate functions involving a total of twelve inputs. It is important to bear in mind that programmable logic provides a convenient means of *implementing* designs. With a real design, some work would be required before this point to conceptualize the design, but due to the simplicity of these circuits, we are already in a position to start the implementation.

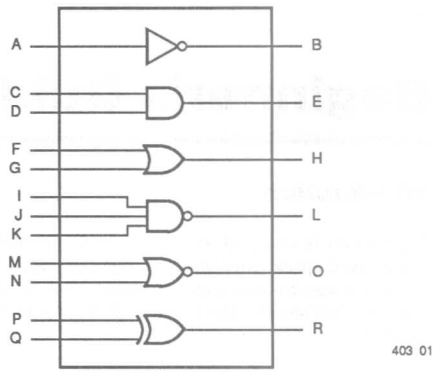


Figure 1. The Basic Logic Gates

Building The Equations

We will start by generating Boolean equations. The first function to be generated is an inverter. This is specified according to Figure 1 as:

```
B = /A
```

Here the "equals" sign ('=') is used to assign a function to output B. The slash ('/') is used to indicate negation, since it is impossible to put a bar over a letter in an ASCII file. Thus this equation may be read:

```
B is TRUE if NOT A is TRUE
```

The next function is a simple AND gate. As shown in Figure 1, we can write

```
E = C*D
```

Here we use the "equals" sign again, but this time we have introduced the asterisk ('*') to indicate the AND operation. This equation may be read:

```
E is TRUE if C AND D are TRUE
```

The third function is an OR gate, which may be written:

```
H = F + G
```

The "plus" sign ('+') is used to specify the OR operation here. Because of the sum-of-products nature of logic as implemented in PLDs, it is often easy to place product terms on separate lines, which improves the readability. We may rewrite this equation as:

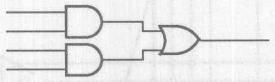
```
H = F
  + G
```

This equation may be read:

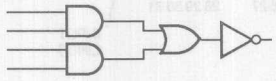
```
H is TRUE if F OR G is TRUE
```

For the moment, we will assume that we have active-HIGH outputs on our device. The functions we have generated so far have essentially been active-HIGH functions. At times we wish to generate active-LOW functions; the next two functions are active-LOW functions that we wish to implement in an active-HIGH device.

When we talk in terms of an active-HIGH or an active-LOW device, the real question is whether there is an extra inverter at the output. An active-HIGH device has an AND-OR structure; an active-LOW device has an AND-OR-INVERT structure which inverts the function at the output (see Figure 2).



a. AND-OR Structure



b. AND-OR-INVERT Structure

403 02

Figure 2. Active HIGH vs. Active LOW

NAND and NOR gates could be generated very simply in an active-LOW device, because we would just have to generate AND and OR functions, and let the output inverter generate their complements. However, given that we wish to implement these functions in an active-HIGH device, we must invoke DeMorgan's theorem, as shown in Figure 3.

```
/(X*Y) = /X + /Y

/(X + Y) = /X*/Y
```

Figure 3. DeMorgan's Theorem

PALASM software has the capability of applying DeMorgan's theorem to functions, so we may generate our NAND function by writing:

```
L = /(I*J*K)
```

or, if preferred,

```
L = /I
  + /J
  + /K
```

Likewise the NOR function may be specified as

```
O = /(M
  +   N)
```

or

```
O = /M*/N
```

Finally, an exclusive-OR (XOR) gate may be specified either as

```
R = P :+: Q
```

where :+: represents the XOR operation, or more explicitly as

```
R = P*/Q
  + /P*Q
```

We have now specified all of the functions in terms of their Boolean equations. The equations are summarized in Figure 4.

```
B = /A

E = C*D

H = F
  + G

L = /I
  + /J
  + /K

O = /M*/N

R = P*/Q
  + /P*Q
```

Figure 4. Basic Gates Equations

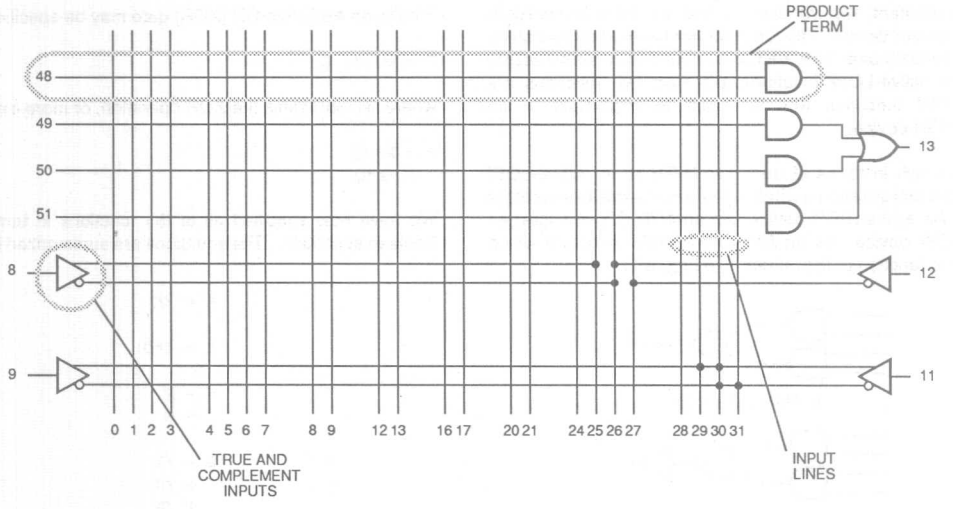
Understanding the Logic Diagram

We will use a PAL12H6 for this function, since it has all of the resources needed to implement all of these functions. A portion of the logic diagram for this device is shown in Figure 5. The full logic diagram for this device can be found on page 5-62.

The logic diagram shows all of the logic resources available in a particular device. In each device, inputs are provided in true and complement versions, as shown in Figure 5. These drive what are often called "input lines", which are the vertical lines in the logic diagram. These input lines can then be connected to product terms. The name "product term" is really just a fancy name for an AND gate. However, PLDs provide very wide gates, which can be cumbersome to draw. To save space, the product terms are drawn as horizontal lines with a small AND gate symbol at one end to indicate the function being performed.

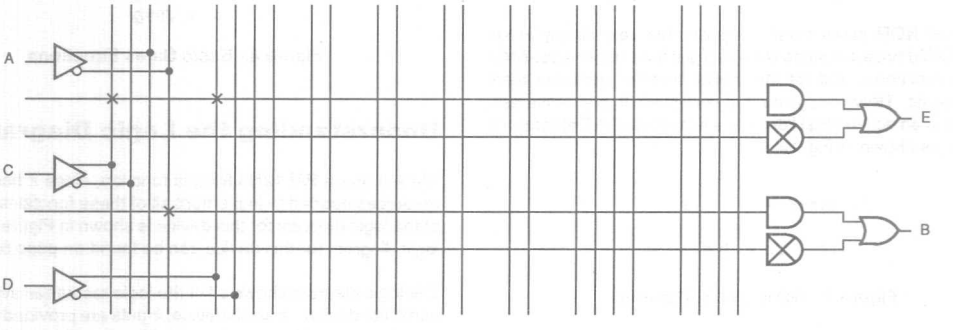
So we see that in the PAL12H6, there are twelve inputs and six outputs. Four of the outputs, on pins 14 through 17, have two product terms connected to the OR gates. The outputs on pins 13 and 18 have four product terms connected to the OR gates. Thus pins 13 and 18 can be used to implement more complex functions than the other four outputs.

Although you really do not need to be concerned with the actual implementation of these functions inside the PAL device, you may be curious. Figure 6 shows how the inverter and the AND gate are implemented in the PAL12H6. An 'X' indicates a connection. A product term that is not used is indicated by an 'X' in the small AND gate.



403 03

Figure 5. A Portion of a Logic Diagram



403 04

Figure 6. Implementation of NOT, AND Gates

Building the Design File

Once the design has been conceptualized, the design file must be generated. This can be done with any editor program. The only requirement is that the editor produce a clean ASCII file. Most word processing programs have this capability, so you can use your favorite program. Any hidden control characters for formatting may cause problems when assembling; thus only "non-document" or "clean" modes should be used. The filename is usually given the extension '.PDS' (for PAL device Design Specification), although this is not required.

We now know exactly what our functions are going to be. We have twelve inputs, six outputs, and the NAND function requires three product terms. Note that if we had specified

```
L = /(I*J*K)
```

instead of

```
L = /I
  + /J
  + /K
```

for the NAND gate, it would not be as obvious how many product terms would be needed.

We are now in a position to create the design file. First you need to enter the PALASM menu. This menu will let you perform all of the tasks you need, including editing your design file. Call up the menu by typing

```
PALASM <CR>
```

When prompted, hit any key to bring the menu onto the screen.

Enter the file name of the design file you wish to create in the field called "Input PDS file". The cursor should be there when the menu appears on the screen. Then hit

```
<CR>
```

This will take you to the filetype field, which should already say ".PDS". We recommend that you use ".PDS" for all of your design files. Hit

```
<CR>
```

again. At this time, the menu will say that the file cannot be opened; this is just because you have not actually created the file yet, so there is no need to be concerned. You can now create your design file by hitting

```
<F3>
```

This will call up your editor, and you can begin building the file.

We start with what is called the DECLARATION SECTION. This section allows you to document your design, and also provides some definitions for the assembler.

The first step is to provide some documentation information. This is done with six keywords:

```
TITLE     (The title of your design goes here)
PATTERN   (The pattern name or number goes here)
REVISION  (The revision number or level goes here)
AUTHOR    (Your name goes here)
COMPANY   (Your company name goes here)
DATE      (The date of creation or modification goes here)
```

This is followed by the CHIP declaration, which tells the assembler which device is being used and what the pin names are going to be. We do this by typing:

```
CHIP    (any name goes here) (the device type goes here)
        (the pin list goes here)
```

Thus, we can create the declaration section for our design as shown in Figure 7.

```
TITLE        Basic gates
PATTERN      P0000
REVISION     A
AUTHOR       Stateyour Namehere
COMPANY      Name of your Co., Inc.
DATE         7/22/87

CHIP  GATES  PAL12H6
A C D F G I J K M GND
N P B E H R O L Q VCC
```

Figure 7. The Declaration Section

Note that the pin list has all pins in order, from pin 1 to pin 20, including VCC and GND. You can see that the inputs were assigned to pins 1–12 and 19, whereas the outputs have been assigned to pins 13–18. Of course these assignments must be made knowing which pins are inputs and outputs, as per the logic diagram. Knowing that the NAND gate requires three product terms, we assigned this function to pin 18, since it can provide up to four product terms.

We can make the pin list easier to read by adding the pin numbers as comments. Anything following a semicolon (;) is treated as a comment, and ignored by the processor. This helps make the design file more readable. The pin list with comments is shown in Figure 8. Notice how much easier it is to read.

```
CHIP  GATES  PAL12H6

;PINS 1  2  3  4  5  6  7  8  9  10
     A  C  D  F  G  I  J  K  M  GND

;PINS 11 12 13 14 15 16 17 18 19 20
     N  P  B  E  H  R  O  L  Q  VCC
```

Figure 8. Pin List with Comments

Next we enter the equations. This section begins with the keyword

```
EQUATIONS
```

after which all of the equations are entered, exactly as shown in Figure 4. Note that the equations may be defined in any order, not necessarily in the order presented in the pin list.

The fact that this is an active-HIGH device does have some bearing on the way the file is built. This is discussed at length on page 6-19 and in the software documentation; we will need to address the issue again below, when we do an active-LOW design. For now, suffice it to say that the design as specified here will work correctly for an active-HIGH device.

Finally, add comments to the equations to document the design. Liberal commenting is encouraged to make it easier for you and others to understand your design in the future.

The equation section, with comments, is shown in Figure 9.

```
EQUATIONS

B = /A                  ;inverter

E = C*D                 ;AND gate

H = F                   ;OR gate
  + G

L = /I                  ;NAND gate after applying
  + /J                  ; DeMorgan's theorem
  + /K

O = /M*/N               ;NOR gate after applying
                        ; DeMorgan's theorem

R = P*/Q                ;XOR gate, expanded
  + /P*Q
```

Figure 9. The Equation Section

When you have completed the file, save the file and get out of the editor.

Generating a JEDEC File

Once the design file has been entered, you can assemble the design to get a JEDEC file. We have two purposes here: to make sure there are no basic mistakes in the file, and to generate a JEDEC file for programming.

To start processing, hit

```
<F5>
```

This gives you a choice of processing options on the right of the screen. To run all of the processing steps automatically, hit

```
6
```

The first program will now run, as evidenced by the file scrolling up the bottom of the screen. If there are any errors, they will appear on the screen and in an error file. These errors will be syntactical in nature; typos, misspelled words, missing declarations, malformed expressions.

Once the first program has been run with no errors, the equations are minimized. It is a good practice to run the expander and minimizer for all designs. Minimization may allow you to fit a design into a smaller device. It also improves the testability of your design. More complicated designs may have to be expanded and minimized.

After the minimization has been performed, the JEDEC file will be generated. If there are any reported errors, it will likely be because you requested some function that the part you chose could not provide. Some examples are:

- Active-HIGH equations in an active-LOW device
- Too many product terms
- Registered equations in a device that has no registers

Since you have used the "autorun" feature, the simulation program will automatically be run. Since we have not specified the simulation yet, this will generate an error. Ignore the error message for now.

When processing is complete, hit

```
<ESC>
```

to get back to the main menu.

If there were any errors during processing, you can see what they were by viewing the run-time log. This is done by hitting

```
<F7>
```

to get a choice of things to view. To view the run-time log, hit

```
1
```

You can scroll down until you find any errors and make a note of what needs correcting. Once you have found the error(s), you can get back to the menu by hitting

```
<ESC><ESC>
```

Any corrections should be made by editing the design file. After making any changes, the design needs to reprocessed.

The JEDEC file will have the same file name as your design file, except that it will have the extension '.JED'. The program also creates an 'xplot' file (with file extension '.XPT') which relates the final design implementation to the logic diagram of the PLD being used. The xplot file is usually not needed, but it confirms the physical implementation of the gates. Figure 10 shows a portion of the xplot for this design; the inverter and the AND gate are shown. You can see how this relates to the logic diagram by referring back to Figure 6.

You may view the complete xplot file by hitting

```
<F7><PgDn><1>
```

To return to the main menu after inspecting the xplot file, hit

```
<ESC>
```

```
40 X--- X--- --OO --OO --OO --OO ---- ----
41 XXXX XXXX XXOO XXOO XXOO XXOO XXXX XXXX
42 0000 0000 0000 0000 0000 0000 0000 0000
43 0000 0000 0000 0000 0000 0000 0000 0000
44 0000 0000 0000 0000 0000 0000 0000 0000
45 0000 0000 0000 0000 0000 0000 0000 0000
46 0000 0000 0000 0000 0000 0000 0000 0000
47 0000 0000 0000 0000 0000 0000 0000 0000

48 ---X ---- --OO --OO --OO --OO ---- ----
49 XXXX XXXX XXOO XXOO XXOO XXOO XXXX XXXX
50 XXXX XXXX XXOO XXOO XXOO XXOO XXXX XXXX
51 XXXX XXXX XXOO XXOO XXOO XXOO XXXX XXXX
52 0000 0000 0000 0000 0000 0000 0000 0000
53 0000 0000 0000 0000 0000 0000 0000 0000
54 0000 0000 0000 0000 0000 0000 0000 0000
55 0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 10. A Portion of an Xplot

Simulating the Gates

After you have verified that your design file is correct, it is time to verify that the design itself is correct. This is done by simulating the design. Simulation provides a way for you to see whether your design is working as you expect it to. You provide a series of commands, or events, which are then simulated by the software. If requested, the software can tell you if the simulation matches what you expect, and, if not, where the problems are.

The simulation section is the last part of the design file. It is not required, but is invariably helpful both in debugging the design, and in generating what can eventually be used as a portion of a test vector sequence. It is introduced by the keyword:

```
SIMULATION
```

The events to be simulated are specified by placing logic levels on pins of interest. Any pins not specifically mentioned will either maintain their present level, or, if no level was ever defined, remain undefined. The SETF command is used to set levels.

We can start by simulating the inverter. To prove that the inverter works as expected, first we wish to set input A HIGH to verify that the output goes LOW. This is done with the statement:

```
SETF A
```

Since all we did here was set A HIGH, all other inputs are still undefined. However, this is enough to determine the output for signal B.

The simulator will calculate the value for B and put it in the output file (to be discussed later), but it is helpful to write what you expect B to be. If the simulator calculates a result that is different from what you expect, it will alert you, and place a marker in the output file to tell you where the problem is. You can do this with the CHECK statement. Since A is HIGH here, we expect B to be LOW. To verify that B is indeed LOW, use the statement:

```
CHECK /B
```

Note that CHECK statements are not required for simulation. However, without them, you must examine the simulation output to see if the design functioned correctly.

We also need to verify that when A goes LOW, B goes HIGH, as expected. This can be done with the pair of statements:

```
SETF /A
CHECK B
```

We have now fully exercised the inverter. However, the standard output file for the simulator, called the history file, tracks all signals on the device, in increasing pin order. From Figure 8, this would be in the order

```
A C D F G I J K M N P B E H R O L Q
```

Instead, we may wish to place related signals near each other. We can do this with the TRACE_ON and TRACE_OFF commands. These commands cause a second output file to be generated, which will only show signals and events as determined by the TRACE_ON command.

In our case, we would like to have the output of the gate appear right after its inputs, so that the workings of the gate can be easily examined. This can be done with the command:

```
TRACE_ON A B C D E F G H I J K L M N O P Q R
```

At the end of the simulation, the TRACE_OFF command is used to end the special trace.

Note that both a history file and a trace file will now be generated. The history file will have the simulation results in increasing pin order; the trace file will have the signals in the order requested with the TRACE_ON command.

Now we can exercise all of the other gates in the circuit. The full simulation section is shown in Figure 11.

```
SIMULATION

TRACE_ON A B C D E F G H I J K L M N O P Q R

;look at the inverter
SETF A                  ;set A HI
CHECK /B                ;verify that B is LO
SETF /A                 ;set A LO
CHECK B                 ;verify that B is HI
                        ;end of inverter trace

;look at AND gate
SETF /C /D              ;set C, D LO
CHECK /E                ;verify E LO
SETF D                  ;C stays LO, D goes HI
CHECK /E                ;E should stay LO
SETF C /D               ;set C HI, D LO
CHECK /E                ;E should stay LO
SETF D                  ;C stays HI, D goes HI
CHECK E                 ;E should now be HI
                        ;end of AND gate trace

;look at OR gate
SETF /F /G              ;set F, G LO
CHECK /H                ;verify H LO
SETF G                  ;F stays LO, G goes HI
CHECK H                 ;H should go HI
SETF F                  ;G stays HI, F goes HI
CHECK H                 ;H should stay HI
SETF /G                 ;F stays HI, G goes LO
CHECK H                 ;H should stay HI
                        ;end of OR gate trace

;look at NAND gate
SETF /I /J /K           ;IJK = 000
CHECK L                 ;verify L HI
SETF K                  ;IJK = 001
CHECK L                 ;L should still be HI
SETF J /K               ;IJK = 010
CHECK L                 ;L should still be HI
SETF K                  ;IJK = 011
CHECK L                 ;L should still be HI
SETF I /J /K            ;IJK = 100
CHECK L                 ;L should still be HI
SETF K                  ;IJK = 101
CHECK L                 ;L should still be HI
SETF J /K               ;IJK = 110
CHECK L                 ;L should still be HI
SETF K                  ;IJK = 111
CHECK /L                ;L should go LO
                        ;end of NAND gate trace

;look at NOR gate
SETF /M /N              ;set M, N LO
CHECK O                 ;verify O HI
SETF N                  ;M stays LO, N goes HI
CHECK /O                ;O should go LO
SETF M                  ;M, N now both HI
CHECK /O                ;O should stay LO
SETF /N                 ;M stays HI, N goes LO
CHECK /O                ;O should stay LO
                        ;end of NOR gate trace

;look at XOR gate
SETF /P /Q              ;set P, Q LO
CHECK /R                ;verify R LO
SETF Q                  ;P stays LO, Q goes HI
CHECK R                 ;R should go HI
SETF P /Q               ;now P HI, Q LO
CHECK R                 ;R should stay HI
SETF Q                  ;both inputs HI
CHECK /R                ;R should go LO

TRACE_OFF               ;end of trace
```

Figure 11. The Simulation Section

This completes the simulation. The entire design file is shown in Figure 21 at the end of this section. You can now run the simulation to make sure that your design is correct.

Running the Simulator

After adding the simulation, the file needs to be reprocessed. Do this again by hitting

<F5>

and then hitting

6

This completely reprocesses the file and runs the simulation.

Each of the events specified in the simulation section of the design file is executed. The resulting outputs are calculated, and, if CHECK statements are used, the calculated outputs are compared to the expected outputs. If the two values do not match, then you are expecting an output that the circuit is not generating. This means either that the expected values are wrong, or that there is a problem with the circuit itself. In either case, the design file must be modified to fix the problem, and then the file should be completely reprocessed.

The main output of the simulator is the history file. In addition, a trace file will be generated if the TRACE_ON statement was used, as in the example above. These files will have the same name as your original design, except that the extension for the history file is '.HST' and that for the trace file is '.TRF'.

You may look at the history and trace files with your editor, or print them out for closer analysis. Note that the history file contains the results of every signal and every event in increasing pin order, whereas the trace file contains signals in the order specifically requested by the TRACE_ON commands.

It is usually easier to look at the simulation output as waveforms. To generate the waveforms, hit

<F7>

To see the waveforms for the trace file, hit

6

Once the waveforms are on the screen, you may scroll around using the arrow keys. The vertical bar cursor will help you line up events.

After you are through examining the file, hit

<ESC>

to get out of the waveform generator, and then,

<ESC>

to get back to the main menu.

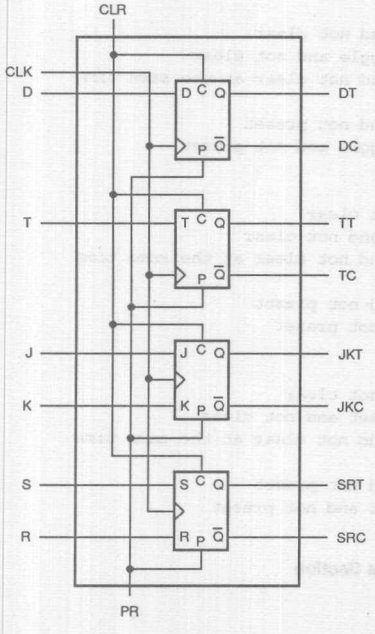
This makes it much easier to debug your design. You can look at the entire simulation in the history file, or break it up into manageable bits in the trace file.

The simulator also converts the simulation results into test vectors, and appends the vectors to the JEDEC file. This gives you two JEDEC files; one with vectors and one without. The new file has the same name as your original design file, but with the extension '.JDC'. This file can be used with programmers that provide functional tests.

Constructing a Registered Design—Basic Flip-Flops

Next we will do a very simple registered design: we will be designing all of the basic flip-flop types (Figure 12). We will conceptualize the design by reviewing briefly the behavior of the D-type flip-flop. We will then present the results for T, J-K, and S-R flip-flops. More detail on these flip-flop types can be found in the logic reference, on page 6-8.

Remember that the devices we will be using only have D-type flip-flops. Thus we will be emulating the other flip-flops with D-type flip-flops.



403 06

Figure 12. Basic Flip-Flops

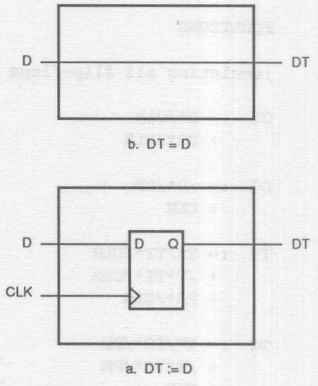
Building the D-Type Flip-Flop Equations

A D-type flip-flop merely presents the input data at the output after being clocked. Its basic transfer function can be expressed as

```
DT := D
```

where we have used pins DT ("D True") and D as shown in Figure 12.

Note the use of ':=' here instead of '='. This indicates that the output is registered for this equation. The difference is illustrated in Figure 13.



403 07

Figure 13. Registered vs. Combinatorial Equations

We can also generate the complement signal (named DC) with the statement

```
DC := /D
```

As per Figure 12, we want to add synchronous preset and clear functions to the flip-flops. This can be done with two input pins, called PR and CLR. To add these functions to the true flip-flop signal, we add /CLR to every product term and add one product term consisting only of PR. Likewise, for the complement functions, we add /PR to every product term, and add one product term consisting only of CLR. This is shown in Figure 14.

```
DT := D*/CLR
    + PR

DC := /D*/PR
    + CLR
```

Figure 14. D-Type Flip-Flops with Clear and Preset

In this way, when clearing the flip-flops, the active-HIGH flip-flops have no product terms true, and go LOW; the active-LOW flip-flops have the last product term true, and will therefore go HIGH. The reverse will occur for the preset function.

There is still one hole in this design: what happens if we preset and clear at the same time? As it is right now, both outputs will go HIGH. This makes no sense since one signal is supposed to be the inverse of the other. To rectify this, we can give the clear function priority over the preset function. We can do this by placing /CLR on every product term for the true flip-flop signal. The results are shown in Figure 15.

```
DT := D*/CLR
    +  PR*/CLR

DC := /D*/PR
    +  CLR
```

Figure 15. D-Type Flip-Flops with Clear Priority

Building the Remaining Equations

The same basic procedure can be applied to all of the other flip-flops. The equations are shown in Figure 16.

```
EQUATIONS

;emulating all flip-flops with D-type flip-flops

DT  := D*/CLR                ;output is D if not clear
     + PR*/CLR               ;or 1 if preset and not clear at the same time

DC  := /D*/PR                ;output is /D if not preset
     + CLR                   ;or 1 if clear

TT  := T*/TT*/CLR            ;go HI if toggle and not clear
     + /T*TT*/CLR            ;stay HI if not toggle and not clear
     + PR*/CLR               ;go HI if preset and not clear at the same time

TC  := T*/TC*/PR             ;go HI if toggle and not preset
     + /T*TC*/PR             ;stay HI if not toggle and not preset
     + CLR                   ;go HI if clearing

JKT := J*/JKT*/CLR           ;go HI if J and not clear
     + /K*JKT*/CLR           ;stay HI if not K and not clear
     + PR*/CLR               ;go HI if preset and not clear at the same time

JKC := /J*/JKC*/PR           ;go HI if not J and not preset
     + K*/JKC*/PR            ;stay HI if K and not preset
     + CLR                   ;go HI if clear

SRT := S*/CLR                ;go HI if set and not clear
     + /R*SRT*/CLR           ;stay HI if not reset and not clear
     + PR*/CLR               ;go HI if preset and not clear at the same time

SRC := R*/PR                 ;go HI if reset and not preset
     + /S*SRC*/PR            ;stay HI if not set and not preset
     + CLR                   ;go HI if clear
```

Figure 16. Flip-Flop Equations Section

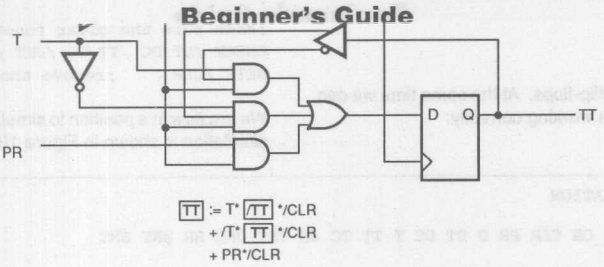


Figure 17. Feedback in the Equation for TT

Notice that in some of the equations above, the output signal itself shows up in the equations. This is the way in which feedback from the flip-flop can be used to determine the next state of the flip-flop. An equivalent logic drawing of the TT equation is shown in Figure 17.

Completing the Design File

We are now in a position to select a device and complete the device file. The device used will be a PAL16R8, which is actually an active-LOW device. The logic diagram for this device can be found on page 5-47. The implications of an active-LOW device are discussed more fully on page 6-19. For now, suffice it to say that one way of implementing active-LOW logic is to declare outputs in the pin list with a slash ('/') in front of them. This essentially defines them as 'negative' or 'inverted' pins.

The declaration section is built up just like the last one, and is shown in Figure 18.

```
TITLE       Basic flip-flops
PATTERN     P0001
REVISION    A
AUTHOR      Stateyour Namehere
COMPANY     Nameofyour Co., Inc.
DATE        7/22/87

CHIP  Flip-flops  PAL16R8

;PINS  1    2  3  4  5   6    7  8  9  10
       CLK  J  K  T  PR  CLR  D  S  R  GND

;PINS  11  12    13    14   15
       OE  /SRC  /SRT  /DC  /DT

;PINS  16   17   18    19    20
       /TC  /TT  /JKC  /JKT  VCC
```

Figure 18. Basic Flip-Flops Declaration Section

Notice that this device has a clock and an output enable pin, called CLK and OE, respectively. The input and output pins are named as shown in Figure 12. The output pins are all defined with slashes, to indicate that they are inverted pins.

The design can now be processed by hitting

<F5>

and then hitting

6

This will tell you whether there are any basic problems with your design. You can correct any mistakes by editing the file and then reprocessing.

Simulating the Flip-Flops

After processing the design and correcting any mistakes, we can write the simulation. We have all of the simulation instructions we need, except for one new instruction which simplifies manipulation of the clock signal. It is possible to use SETF instructions with the clock pin, but then each clock transition would require two instructions: one to set the clock HIGH, and one to bring it back LOW.

Instead, we can use the CLOCKF instruction. This pulses a clock pin in one instruction. Of course, registered outputs will not change state until after the rising edge of the clock signal. Since we have named the clock pin CLK, we can clock this device with the instruction

```
CLOCKF CLK
```

We can simulate the true D-type flip-flop as follows:

```
SETF D          ;set the D input HI
CLOCKF CLK      ;clock the device
CHECK DT        ;verify that the output went HI
SETF /D         ;set the D input LO
CLOCKF CLK      ;clock the device
CHECK /DT       ;verify that the output went LO
```

Before we actually simulate any registered design like this, two items must be initialized: the clock and the output enable pin (OE). Since the CLOCKF statement executes a HIGH-LOW pulse on the clock pin, we must first make sure that the clock is set LOW

to begin with. We must also enable the outputs, which is done by setting the OE pin LOW. These tasks can be accomplished with the statement:

```
SETF /CLK /OE
```

We also need to initialize the flip-flops. At the same time we can verify that the CLR function is working correctly:

```
SETF CLR /PR        ;set the clear pin
CLOCKF CLK          ;clear the circuit

;make sure the clear function worked
CHECK /DT DC /TT TC /JKT JKC /SRT SRC
SETF /CLR       ;remove the clear signal
```

We are now in a position to simulate the entire circuit. A thorough simulation is shown in Figure 19.

```
SIMULATION

TRACE_ON CLR PR D DT DC T TT TC JK JKT JKC SR SRT SRC

;initialize the circuit
SETF CLR /PR                ;set the clear pin
CLOCKF CLK                  ;clear the circuit

;make sure the clear function worked
CHECK /DT DC /TT TC /JKT JKC /SRT SRC
SETF /CLR                   ;remove the clear signal

;check out the preset function
SETF PR                     ;set the preset pin
CLOCKF CLK                  ;preset the circuit
CHECK DT /DC TT /TC JKT /JKC SRT /SRT
SETF /PR                    ;remove the preset signal

;verify that clear has priority
SETF PR CLR                 ;set both clear and preset
CLOCKF CLK
CHECK /DT DC /TT TC /JKT JKC /SRT SRC
SETF /PR /CLR               ;remove both preset, clear

;disable all flip-flop inputs for now
SETF /D /T /J /K /S /R

;check out D-type flip-flops
SETF D                      ;set the D input HI from LO
CLOCKF CLK                  ;clock the device
CHECK DT /DC                ;verify that the true output went HI
                            ;and the complement went LO

SETF D                      ;hold a HI
CLOCKF CLK
CHECK DT /DC                ;verify that state maintained

SETF /D                     ;set the D input LO from HI
CLOCKF CLK                  ;clock the device
CHECK /DT DC                ;verify that the true output went LO
                            ;and complement went HI

SETF /D                     ;hold a LO
CLOCKF CLK
CHECK /DT DC                ;verify that state maintained


;check out T-type flip-flops
SETF /T                     ;hold a LO
CLOCKF CLK
CHECK /TT TC                ;true output should still be LO
                            ;complement output HI

SETF T                      ;toggle from a LO
CLOCKF CLK
CHECK TT /TC                ;both outputs should have changed state
```

Figure 19. Flip-Flop Simulation Section

```
SETF /T                    ;hold a HI
CLOCKF CLK
CHECK TT /TC               ;both outputs should have held their state

SETF T                     ;toggle from a HI
CLOCKF CLK
CHECK /TT TC               ;both outputs should have changed state

;check out J-K flip-flops
SETF /J /K                 ;hold a LO
CLOCKF CLK
CHECK /JKT JKC             ;both outputs should have held their state

SETF J /K                  ;set a HI
CLOCKF CLK
CHECK JKT /JKC             ;outputs should have changed

SETF /J /K                 ;hold a HI
CLOCKF CLK
CHECK JKT /JKC             ;both outputs should have held their state

SETF /J K                  ;reset a LO
CLOCKF CLK
CHECK /JKT JKC             ;outputs should have changed

CLOCKF CLK                 ;reset an output that is already LO
CHECK /JKT JKC             ;make sure that the outputs didn't change

SETF J K                   ;toggle from a LO
CLOCKF CLK
CHECK JKT /JKC             ;verify that outputs changed

SETF J /K                  ;set an output that is already HI
CLOCKF CLK
CHECK JKT /JKC             ;make sure the outputs didn't change

SETF J K                   ;toggle from a HI
CLOCKF CLK
CHECK /JKT JKC             ;verify that outputs changed


;check out S-R flip-flops
SETF /S /R                 ;hold a LO
CLOCKF CLK
CHECK /SRT SRC             ;both outputs should have held their state

SETF S /R                  ;set a HI
CLOCKF CLK
CHECK SRT /SRC             ;outputs should have changed

SETF /S /R                 ;hold a HI
CLOCKF CLK
CHECK SRT /SRC             ;both outputs should have held their state

SETF S /R                  ;set an output that is already HI
CLOCKF CLK
CHECK SRT /SRC             ;make sure the outputs didn't change

SETF /S R                  ;reset a LO
CLOCKF CLK
CHECK /SRT SRC             ;outputs should have changed

CLOCKF CLK                 ;reset an output that is already LO
CHECK /SRT SRC             ;make sure that the outputs didn't change

TRACE_OFF
```

Figure 19. Flip-Flop Simulation Section (Cont'd.)

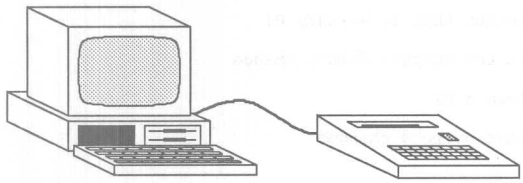
The file can now be simulated in the same manner as the basic gates design. The simulation results can be viewed either by examining the history or trace file, or by generating waveforms. The complete design file is shown in Figure 22 at the end of this section.

Programming a Device

After simulating the design, and verifying that it works, it is time to program a device. There are several steps to programming, but the exact operation of the programmer naturally depends on the type of programmer being used. We will be as explicit as we can here, but you will need to refer to your programmer manual for the specifics.

The first thing that must be done after turning the programmer on is to select the device type. This tells the programmer what kind of programming data to expect. The device type is usually selected either from a menu or by entering a device code. Your programmer manual will have the details.

Next a JEDEC file must be downloaded. To transfer the JEDEC file from the computer to your programmer, you will need to provide a connection, as shown in Figure 20. This is normally done with an RS-232 cable connected between the programmer's port and a serial port on the computer (usually COM1).



403 09

Figure 20. A Connector Must Be Provided Between the Computer and the Programmer

If your programmer can perform functional tests, and you wish for those tests to be performed, you should download the '.JDC' file; otherwise you should download the '.JED' file.

To download data, the programmer must first be set up to receive data. The programmer manual will tell you how to do this. The data can then be sent from the computer by hitting

<F4>

This sets up communication between the computer and the programmer. Whichever communication program is installed will be invoked. This is used to transmit the JEDEC file to the programmer. The details below assume you are using PC2; if not, follow the instructions for your program to accomplish the same steps.

Before actually sending the data, you must verify the correct communication protocol. Check to make sure you know what protocol the programmer is expecting; then hit the <F2> key on the computer. This allows you to set up the baud rate, data bits, stop bits, and parity.

Once the protocol has been set up, hit the <F1> key. You will be asked for the file name; enter the name of the JEDEC file you wish to use. The computer will then announce that it is sending the data, and tell you when it is finished. Note that just because it says it has finished sending data does not mean that the data was received. Your programmer will indicate whether or not data was received correctly.

Once the data has been received, the programmer is ready to program a device. Place a device in the appropriate socket, and follow the instructions for your programmer to program the device. This procedure programs and verifies the connections in the device, and, if a '.JDC' file was used, will perform a functional test.

The programmer will announce when the programming procedure has been completed. You may then take the device and plug it into your application.

If you have actually programmed one of the examples that we created above, you naturally don't have a board into which you can plug the device. If you do have a lab setup, you may wish to play with the devices to verify for yourself that the devices perform just as you expected them to.

You will find much more detail on many issues that were not discussed in this section in the remaining sections of this handbook. This section should have provided you with the basic knowledge you need to understand the remaining design examples in this book, and to start your own designs.

```
TITLE          Basic gates
PATTERN        P0000
REVISION       A
AUTHOR         Stateyour Namehere
COMPANY        Nameofyour Co., Inc.
DATE           7/22/87

CHIP  GATES  PAL12H6

;PINS  1  2  3  4  5  6  7  8  9  10
       A  C  D  F  G  I  J  K  M  GND

;PINS  11 12 13 14 15 16 17 18 19 20
       N  P  B  E  H  R  O  L  Q  VCC

EQUATIONS

B = /A                          ;inverter

E = C*D                         ;AND gate

H = F                           ;OR gate
  + G

L = /I                          ;NAND gate after applying DeMorgan's theorem
  + /J
  + /K

O = /M*/N                       ;NOR gate after applying DeMorgan's theorem

R = P*/Q                        ;XOR gate, expanded
  + /P*Q

SIMULATION

TRACE_ON A B C D E F G H I J K L M N O P Q R

;look at the inverter
SETF A                          ;set A HI
CHECK /B                        ;verify that B is LO
SETF /A                         ;set A LO
CHECK B                         ;verify that B is HI
                                ;end of inverter trace

;look at AND gate
SETF /C /D                      ;set C, D LO
CHECK /E                        ;verify E LO
SETF D                          ;C stays LO, D goes HI
CHECK /E                        ;E should stay LO
SETF C /D                       ;set C HI, D LO
CHECK /E                        ;E should stay LO
SETF D                          ;C stays HI, D goes HI
CHECK E                         ;E should now be HI
                                ;end of AND gate trace

;look at OR gate
SETF /F /G                      ;set F, G LO
CHECK /H                        ;verify H LO
SETF G                          ;F stays LO, G goes HI
CHECK H                         ;H should go HI
SETF F                          ;G stays HI, F goes HI
CHECK H                         ;H should stay HI
SETF /G                         ;F stays HI, G goes LO
CHECK H                         ;H should stay HI
                                ;end of OR gate trace
```

Figure 21. Complete Basic Gates Design File

```
;look at NAND gate
SETF /I /J /K              ;IJK = 000
CHECK L                    ;verify L HI
SETF K                     ;IJK = 001
CHECK L                    ;L should still be HI
SETF J /K                  ;IJK = 010
CHECK L                    ;L should still be HI
SETF K                     ;IJK = 011
CHECK L                    ;L should still be HI
SETF I /J /K               ;IJK = 100
CHECK L                    ;L should still be HI
SETF K                     ;IJK = 101
CHECK L                    ;L should still be HI
SETF J /K                  ;IJK = 110
CHECK L                    ;L should still be HI
SETF K                     ;IJK = 111
CHECK /L                   ;L should go LO
                           ;end of NAND gate trace

;look at NOR gate
SETF /M /N                 ;set M, N LO
CHECK O                    ;verify O HI
SETF N                     ;M stays LO, N goes HI
CHECK /O                   ;O should go LO
SETF M                     ;M, N now both HI
CHECK /O                   ;O should stay LO
SETF /N                    ;M stays HI, N goes LO
CHECK /O                   ;O should stay LO
                           ;end of NOR gate trace

;look at XOR gate
SETF /P /Q                 ;set P, Q LO
CHECK /R                   ;verify R LO
SETF Q                     ;P stays LO, Q goes HI
CHECK R                    ;R should go HI
SETF P /Q                  ;now P HI, Q LO
CHECK R                    ;R should stay HI
SETF Q                     ;both inputs HI
CHECK /R                   ;R should go LO

TRACE_OFF                  ;end of trace
```

Figure 21. Complete Basic Gates Design File (Cont'd.)

```
TITLE          Basic flip-flops
PATTERN        P0001
REVISION       A
AUTHOR         Stateyour Namehere
COMPANY        Nameofyour Co., Inc.
DATE           7/22/87

CHIP  Flip-flo  PAL16R8

;PINS 1    2  3  4  5   6    7  8  9  10
      CLK  J  K  T  PR  CLR  D  S  R  GND

;PINS 11  12    13    14   15
      OE  /SRC  /SRT  /DC  /DT

;PINS 16   17   18    19    20
      /TC  /TT  /JKC  /JKT  VCC

EQUATIONS

;emulating all flip-flops with D-type flip-flops

DT := D*/CLR                ;output is D if not clear
    + PR*/CLR               ;or 1 if preset and not clear at the same time


DC := /D*/PR                ;output is /D if not preset
    + CLR                   ;or 1 if clear


TT := T*/TT*/CLR            ;go HI if toggle and not clear
    + /T*TT*/CLR            ;stay HI if not toggle and not clear
    + PR*/CLR               ;go HI if preset and not clear at the same time


TC := T*/TC*/PR             ;go HI if toggle and not preset
    + /T*TC*/PR             ;stay HI if not toggle and not preset
    + CLR                   ;go HI if clearing


JKT := J*/JKT*/CLR          ;go HI if J and not clear
    + /K*JKT*/CLR           ;stay HI if not K and not clear
    + PR*/CLR               ;go HI if preset and not clear at the same time


JKC := /J*/JKC*/PR          ;go HI if not J and not preset
     + K*JKC*/PR            ;stay HI if K and not preset
     + CLR                  ;go HI if clear


SRT := S*/CLR               ;go HI if set and not clear
     + /R*SRT*/CLR          ;stay HI if not reset and not clear
     + PR*/CLR              ;go HI if preset and not clear at the same time


SRC := R*/PR                ;go HI if reset and not preset
     + /S*SRC*/PR           ;stay HI if not set and not preset
     + CLR                  ;go HI if clear
```

Figure 22. Complete Basic Flip-Flops Design File

```
SIMULATION

TRACE_ON CLR PR D DT DC T TT TC JK JKT JKC SR SRT SRC

;initialize the circuit
SETF CLR /PR                ;set the clear pin
CLOCKF CLK                  ;clear the circuit

;make sure the clear function worked
CHECK /DT DC /TT TC /JKT JKC /SRT SRC
SETF /CLR                   ;remove the clear signal

;check out the preset function
SETF PR                     ;set the preset pin
CLOCKF CLK                  ;preset the circuit
CHECK DT /DC TT /TC JKT /JKC SRT /SRT
SETF /PR                    ;remove the preset signal

;verify that clear has priority
SETF PR CLR                 ;set both clear and preset
CLOCKF CLK
CHECK /DT DC /TT TC /JKT JKC /SRT SRC
SETF /PR /CLR               ;remove both preset, clear

;disable all flip-flop inputs for now
SETF /D /T /J /K /S /R

;check out D-type flip-flops
SETF D                      ;set the D input HI from LO
CLOCKF CLK                  ;clock the device
CHECK DT /DC                ;verify that the true output went HI and the complement went LO

SETF D                      ;hold a HI
CLOCKF CLK
CHECK DT /DC                ;verify that state maintained

SETF /D                     ;set the D input LO from HI
CLOCKF CLK                  ;clock the device
CHECK /DT DC                ;verify that the true output went LO and complement went HI

SETF /D                     ;hold a LO
CLOCKF CLK
CHECK /DT DC                ;verify that state maintained


;check out T-type flip-flops
SETF /T                     ;hold a LO
CLOCKF CLK
CHECK /TT TC                ;true output should still be LO, complement output HI

SETF T                      ;toggle from a LO
CLOCKF CLK
CHECK TT /TC                ;both outputs should have changed state

SETF /T                     ;hold a HI
CLOCKF CLK
CHECK TT /TC                ;both outputs should have held their state

SETF T                      ;toggle from a HI
CLOCKF CLK
CHECK /TT TC                ;both outputs should have changed state
```

Figure 22. Complete Basic Flip-Flops Design File (Cont'd.)

```
;check out J-K flip-flops
SETF /J /K                  ;hold a LO
CLOCKF CLK
CHECK /JKT JKC              ;both outputs should have held their state

SETF J /K                   ;set a HI
CLOCKF CLK
CHECK JKT /JKC              ;outputs should have changed

SETF /J /K                  ;hold a HI
CLOCKF CLK
CHECK JKT /JKC              ;both outputs should have held their state

SETF /J K                   ;reset a LO
CLOCKF CLK
CHECK /JKT JKC              ;outputs should have changed

CLOCKF CLK                  ;reset an output that is already LO
CHECK /JKT JKC              ;make sure that the outputs didn't change

SETF J K                    ;toggle from a LO
CLOCKF CLK
CHECK JKT /JKC              ;verify that outputs changed

SETF J /K                   ;set an output that is already HI
CLOCKF CLK
CHECK JKT /JKC              ;make sure the outputs didn't change

SETF J K                    ;toggle from a HI
CLOCKF CLK
CHECK /JKT JKC              ;verify that outputs changed


;check out S-R flip-flops
SETF /S /R                  ;hold a LO
CLOCKF CLK
CHECK /SRT SRC              ;both outputs should have held their state

SETF S /R                   ;set a HI
CLOCKF CLK
CHECK SRT /SRC              ;outputs should have changed

SETF /S /R                  ;hold a HI
CLOCKF CLK
CHECK SRT /SRC              ;both outputs should have held their state

SETF S /R                   ;set an output that is already HI
CLOCKF CLK
CHECK SRT /SRC              ;make sure the outputs didn't change

SETF /S R                   ;reset a LO
CLOCKF CLK
CHECK /SRT SRC              ;outputs should have changed

CLOCKF CLK                  ;reset an output that is already LO
CHECK /SRT SRC              ;make sure that the outputs didn't change

TRACE_OFF
```

Figure 22. Complete Basic Flip-Flops Design File (Cont'd.)

IMPORTANT
NOTES

PLD Design Methodology

Programmable logic devices (PLDs) are used in digital systems design for implementing a wide variety of logic functions. These logic functions range from simple random logic replacement (discussed earlier in the Beginner's Guide to PLDs on page 2-2) to complex control sequencers (discussed in the subsequent sections). Programmable logic devices offer the multiple advantages of low cost, high integration, ease of use, and easier design debugging capability not available in other systems design options. These are described in the introduction (on page 1-1) to this handbook. In the following discussion we will detail the PLD design process.

Most PLDs have an AND-OR array structure with programmable connections in either or both of the arrays. A programmable array implies that the connections can be programmed by the user. PLDs are classified depending on which of the two arrays is programmable or fixed. The popular PAL (Programmable Array Logic) devices have a programmable AND array and a fixed OR array. PAL devices are used for a wide variety of combinatorial and registered logic functions. PROM (Programmable Read-Only Memory) devices—used often as memory and seldom as logic—have a fixed AND array and a programmable OR array. Both arrays are programmable in PLS (Programmable Logic Sequencer) devices. These devices are used for special state machine applications discussed on page 2-117. In addition, there are other programmable logic devices that combine programmable arrays with dedicated logic for providing optimal functionality for a specific system application. The PROSE (PROgrammable SEquencer) device is one such device, with its architecture optimized for state machine designs. A discussion of the various PLD architectures is included on page 1-7. In this discussion we will also examine the various design constraints to be considered when selecting the correct architecture for a given application.

All digital logic can be efficiently reduced to two fundamental gates, AND and OR, provided both true and complement versions of all input signals are available. Such logic is generally built around what is known as the sum-of-products (AND-OR) form. Please see page 6-1 for details on sum-of-products form and Boolean logic. Programmable logic devices are ideal for implementing such two-stage logic in the AND and OR arrays.

Various process technologies offer many design options for PLDs. The connections in the programmable arrays can be fuse-based, commonly used in both ECL and TTL bipolar technologies, E/EEPROM cell-based in UV-EPROM and EEPROM CMOS technologies and RAM cell-based in CMOS RAM technology. ECL PLDs are used for very high-speed designs (greater than 125 MHz bandwidth), while CMOS is used for low-power designs. Bipolar TTL fuse-based PLDs cover the vast midrange of applications and are the most popular PLDs. The selection of technology is mostly dependent upon the system speed and power constraints. Most design engineers are familiar with these constraints, which not only dictate the technology of PLDs but of all other logic used in a system. In the following discussion, we will assume TTL fuse-based technology for simplicity. A detailed discussion of the technology is included on page 3-130.

Designing with PLDs involves the use of design software and a device programmer (Figure 1). The design software eliminates the need to identify every connection to be programmed for implementing the desired sum-of-products logic. The design process begins with the creation of a design file which specifies the desired function. The function is typically represented by its sum-of-products form and can be derived directly from the timing diagram and/or truth tables. Occasionally Karnaugh maps and state diagrams are also used. The design file is then assembled to produce the "JEDEC" file. The JEDEC file gets its name from the fact that it is an approved JEDEC standard for specifying the state of every connection on the device. Simulation can then be performed. If the design is correct, the JEDEC file is downloaded into a device programmer for programming the connections on the device. The device can then be plugged into the PC board where it will function. The entire procedure can often be performed with the designer never having to leave the desk. Most programmers interface to personal computers, so that the design file can be edited, assembled, simulated, and downloaded, and the device programmed, all in one place.

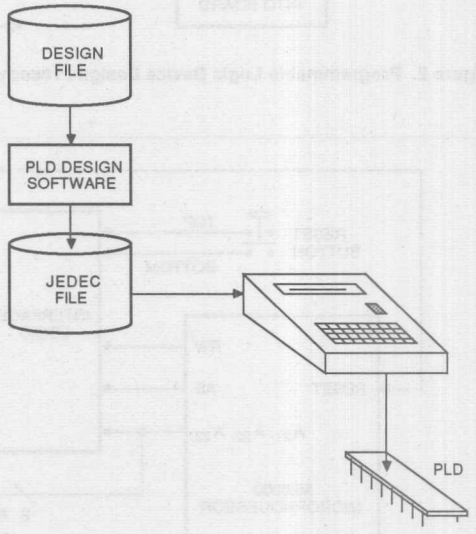


Figure 1. PLDs are Designed using Software and a Device Programmer

The first stage in a PLD design process (Figure 2) is the conceptualization of a design problem; the second is the selection of the correct device; the third is the implementation of the design, which also includes simulating the design with test vectors; and finally, the actual programming and testing on a system board. We will take a simple design example and go through the various stages of this design process.

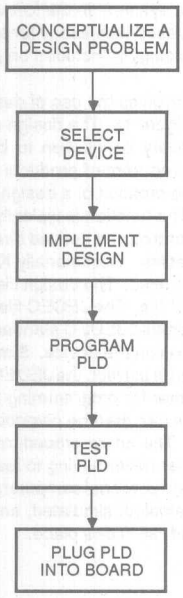


Figure 2. Programmable Logic Device Design Process

Conceptualizing a Design

The first step in the PLD design process is also required for any SSI/MSI design. An advantage of PLDs is that at this stage the designer needs to be concerned only with the required logic function. With SSI or MSI, various device logic limitations must be accounted for before the design can be started. Clearly a designer needs to develop a brief and complete functional description, based upon the system design requirements.

We will take the example of a simple address decoder circuit required for a 68000 microprocessor. The microprocessor has 24 address lines along with separate read and write signals. It requires some ROM to store the boot-up code as well as some RAM for storing and executing programs. The purpose of the address decoder circuitry is to select one of the memory addresses at a time. The RAMs and ROMs are assigned addresses on the 68000 microprocessor address space. The address decoder circuit has to select one of the RAMs or ROMs for a specific range of addresses, called the address space. This selection is accomplished by asserting the specific chip-select signal for the RAM or ROM when the microprocessor accesses one of the addresses in the address space. There is additional circuitry in a typical microprocessor system for addressing I/O

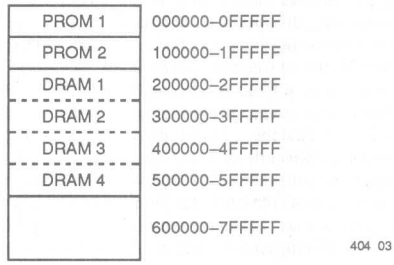


Figure 3. Memory Address Map

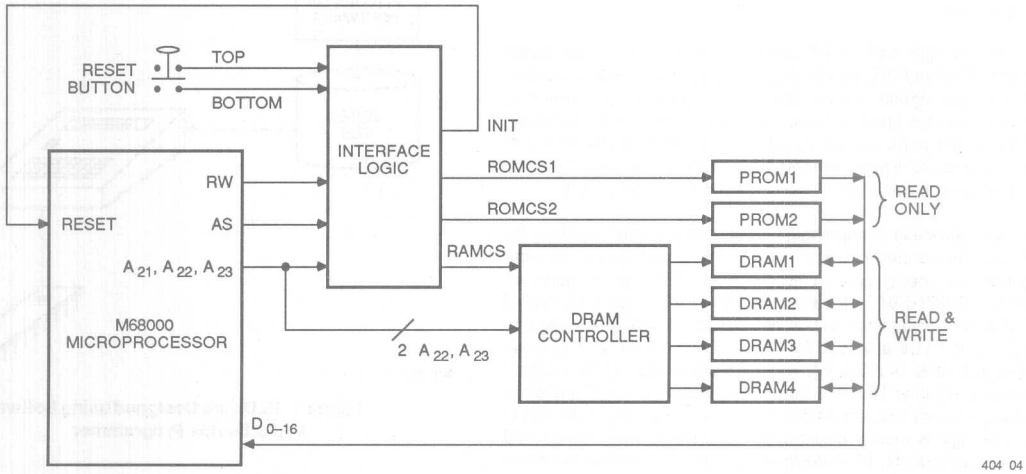


Figure 4. Microprocessor to Memory Interface

devices (such as disk controllers). These devices also require that chip-select signals be asserted when the microprocessor addresses them. Figure 3 shows an example address map for a 68000 microprocessor.

Figure 4 shows the circuit diagram. The address signals from the 68000 microprocessor are inputs to the interface logic block. The outputs generated are ROMCS1, ROMCS2, and RAMCS. The generation of signals for selecting device I/Os is similar and is not shown here for the sake of simplicity. Other system inputs to the interface are the address strobe signal generated by the 68000 microprocessor as well as the read/write signal. The truth table for generating the outputs is shown in Figure 5. This truth table is derived from the memory address map and the functional description of the design.

ADDRESSES HEX	SIZE	A23	A22	A21	SIGNAL
00000-0FFFF	1Mbytes	0	0	0	$\overline{\text{ROMCS1}}$
10000-1FFFF	1Mbytes	0	0	1	$\overline{\text{ROMCS2}}$
20000-2FFFF	1Mbytes	0	1	0	$\overline{\text{RAMCS}}$
30000-3FFFF	1Mbytes	0	1	1	$\overline{\text{RAMCS}}$
40000-4FFFF	1Mbytes	1	0	0	$\overline{\text{RAMCS}}$
50000-5FFFF	1Mbytes	1	0	1	$\overline{\text{RAMCS}}$

Figure 5. Truth Table for Chip-Select Signals

Device Selection Considerations

The first task for the designer is to identify the design problem and classify it as a combinatorial function or a registered function, depending upon whether or not registers are required. In most cases, this decision depends upon the functional nature of the problem. Sometimes timing and logic considerations can also dictate the use of registers; this will be discussed later. Registers are usually not required for such simple combinatorial functions such as encoders, decoders, multiplexers, demultiplexers, adders, and comparators. Registers are required, however, for functions such as counters, timers, control signal generation, and state machines. No registers are required for this simple address decoding example.

A brief look at the places where various architectural features are used will help us select a device for our design example. Based on historic record of usage and architectural functionality, PLDs can be classified into three basic application segments. Figure 6 shows the optimal placement of PAL, PLS, and PROSE/FPC (Fuse Programmable Controller) devices for these three segments; combinatorial designs, simple registered designs and complex registered (sequencer) designs. This classification also shows the available functionality and device speeds. These are rough estimates that most designers implicitly assume and develop with experience. These estimates vary drastically among different designers, based on personal preferences and experience. They have been included here to provide a complete road map for device selection and should be considered strictly as a rule of thumb.

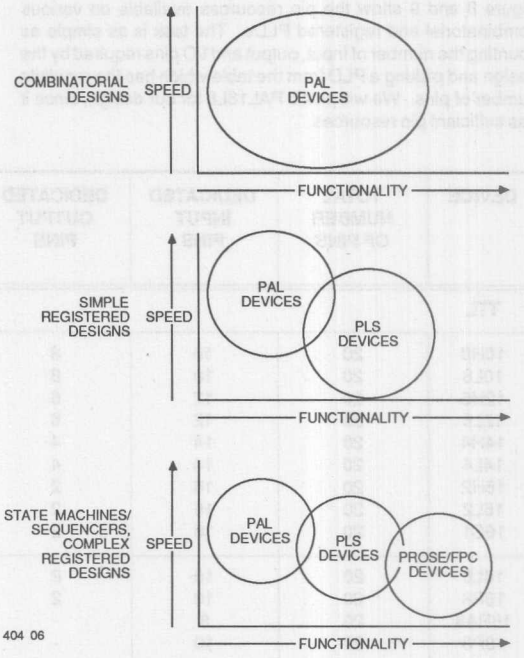


Figure 6. Various PLD Design Options

Based on these criteria, the best choice for our combinatorial design would be a PAL device. The task now is to select a PAL device for implementing the desired function. The main function-specific selection considerations are shown in Figure 7. These items are applicable to most designs. There are other function-specific issues which need addressing when selecting a device; these will be discussed in the later sections of the book, when we address advanced designs.

1. **Number of input pins**
2. **Number of output pins**
3. **Number of I/O pins**
4. **Device speed**
5. **Device power requirements**
6. **Number of registers (if any)**
7. **Number of product terms**
8. **Output polarity control**

Figure 7. General Device Selection Considerations

The first resource that must be provided in a PLD is the number of pins needed for the basic logic function. This consists of the number of input and output pins. Many PLDs have internal feedback, which allows the generated output signal to be reused as an input. The same feedback also allows the pin to be used as a dedicated input, if required. This is especially useful for fitting various designs with different input/output requirements on the same device. The I/O pin capability of certain PLDs can also be very useful for certain bus applications and will be discussed later.

Figure 8 and 9 show the pin resources available on various combinatorial and registered PLDs. The task is as simple as counting the number of input, output and I/O pins required by the design and picking a PLD from the table which has the requisite number of pins. We will pick a PAL16L8 for our design, since it has sufficient pin resources.

The next selection issue is the device speed. The most important timing consideration for combinatorial PLDs is the propagation delay (t_{PD}) of signals from the input to the output of the device. For registered PLDs, the important timing consideration is the device clocking frequency. This clocking frequency (shown in Figure 9) is in turn determined by sum of the register setup time (t_{su}), and clock-to-output propagation delay (t_{CLK}). Most systems impose

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES t_{PD} IN ns
TTL						
10H8	20	10	8	-	2	35
10L8	20	10	8	-	2	35
12H6	20	12	6	-	2-4	35
12L6	20	12	6	-	2-4	35
14H4	20	14	4	-	4	35
14L4	20	14	4	-	4	35
16H2	20	16	2	-	8	35
16L2	20	16	2	-	8	35
16C1	20	16	2	-	16	40
16L8	20	10	2	6	7	10, 15, 25, 35
16P8	20	10	2	6	7	25
16RA8	20	8	-	8	4	20, 30
18P8	20	10	-	8	8	15, 25
6L16	24	6	16	-	1	25
8L14	24	8	14	-	1	25
12L10	24	12	10	-	2	40
14L8	24	14	8	-	2-4	40
16L6	24	16	6	-	2-4	40
18L4	24	18	4	-	4-6	40
20L2	24	20	2	-	8	40
20C1	24	20	2	-	16	40
20L8	24	14	2	6	7	15, 25, 35
20L10	24	12	2	8	3	15, 20, 25, 35
20RA10	24	10	-	10	4	20, 30
20S10	24	12	2	8	8-16	35
22P10	24	12	-	10	8	15, 25
22XP10	24	12	-	10	8	20, 30, 40
22RX8	24	14	-	8	8	25
22V10	24	12	-	10	8-16	15, 25, 35
32VX10	24	12	-	10	8-16	25, 30
CMOS						
C16L8	20	10	2	6	7	25
C20L8	24	14	2	6	7	35, 45
C22V10	24	12	-	10	8-16	25, 35
C29M16	24	5	-	16	8-16	35, 45
C29MA16	24	5	-	16	4-12	35, 45
ECL						
10H20P8	24	12	-	8	4-8	6

Figure 8. Combinatorial PLDs with Pin Resources and Speed Requirements

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16R8	20	8	8	-	8	55.5, 37, 25, 16
16R6	20	8	6	2	8	55.5, 37, 25, 16
16R4	20	8	4	4	8	55.5, 37, 25, 16
16RP8	20	8	8	-	8	22.2
16RP6	20	8	6	2	8	22.2
16RP4	20	8	4	4	8	22.2
16RA8	20	8	0-8	8-0	4	20
16X4	20	8	4	4	8	14
23S8	20	9	4	4	8-12	33.3, 28.5
20R8	24	12	8	-	8	37, 25, 16
20R6	24	12	6	2	8	37, 25, 16
20R4	24	12	4	4	8	37, 25, 16
20X10	24	10	10	-	4	22.2
20X8	24	10	8	2	4	22.2
20X4	24	10	4	6	4	22.2
20RP10	24	10	10	-	8	37, 25
20RP8	24	10	8	2	8	37, 25
20RP6	24	10	6	4	8	37, 25
20RP4	24	10	4	6	8	37, 25
20XRP10	24	10	10	-	8	30, 22.2, 14
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
20RS10	24	10	10	-	8-16	19.2
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
32R16	40	16	16	-	8-16	16
CMOS						
C16R8	20	8	8	-	8	28.5
C16R6	20	8	6	2	8	28.5
C16R4	20	8	4	4	8	28.5
C20R8	24	12	8	-	8	20,15.3
C20R6	24	12	6	2	8	20,15.3
C20R4	24	12	4	4	8	20,15.3
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
10H/020EV/G8	24	12	0-8	8-0	8-12	125
10H20G8 (Latched)	24	12	0-8	8-0	4-8	N.A.

Figure 9. Registered PLDs with Pin Resources and Speed Requirements

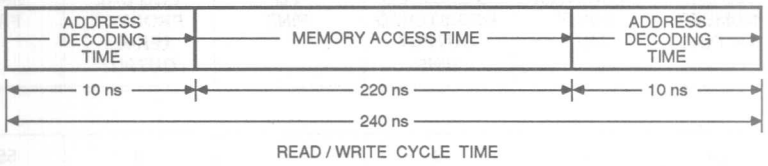


Figure 10. System Timing Requirements

some timing restrictions on the internal logic functions. These restrictions will determine the necessary t_{PD} (for combinatorial devices) or f_{MAX} (for registered devices). Details on clocking frequency are provided on page 2-64.

In our design example, the PLD will primarily perform address decoding. The critical system timing constraint is determined by the read/write cycle time of the microprocessor and the memory access time available (Figure 10). Most microprocessors allow anywhere from 10 to 35 ns for address decoding. That is, 10–35 ns after the address is available, the correct memory chip-select signal should be asserted. In our design example the available cycle time of 240 ns and memory access time of 220 ns leaves barely 10 ns for address decode time. From the table in Figure 8, we can check the propagation delay and select the appropriate speed of PAL16L8 for our design, which is t_{PD}=10 ns.

We have already briefly discussed the types of applications where registers are needed. Sometimes the consideration of system timing can affect whether or not registers are needed. Devices with registers can hold a signal stable for the long durations required by the addressed peripheral or memory. However, this slows the initial response or access time of the device since the chip select must wait for the setup time before the rising edge of the clock cycle. Devices without registers provide fast access time but hold the signal valid only as long as the input conditions are valid. In most address decoders, the address signals are kept asserted by the microprocessor until the read/write cycle is completed. In this case, the registers are not required for holding the signals asserted.

The remaining two general design considerations are the number of product terms and output polarity. We will discuss these two as we implement the design in the next section.

Implementing a design

Implementing a design (Figure 11) requires the creation of a design file. This design file is called the PAL device Design Specification (PDS) in the PALASM 2 software package. Details of PALASM 2 software can be found in section 4 of this handbook. The design file contains three types of information.

1. Basic bookkeeping information
2. Design syntax
3. Simulation syntax

The first section provides documentation information, along with the name of the device selected, and the signal names. All of the signal names are assigned to pins and are defined as inputs or outputs. The Declaration Section contains the signal names in the ascending order of pin numbers. Figure 12 shows the pin assignments of the signals used in our example. Signals which are used as I/Os are assigned as outputs and are used as both inputs and outputs in the sum-of-products logic explained later.

Once the design file is complete, it is then assembled and simulated. Once it passes assembly and simulation, the resultant JEDEC file is downloaded to a device programmer for configuring the device.

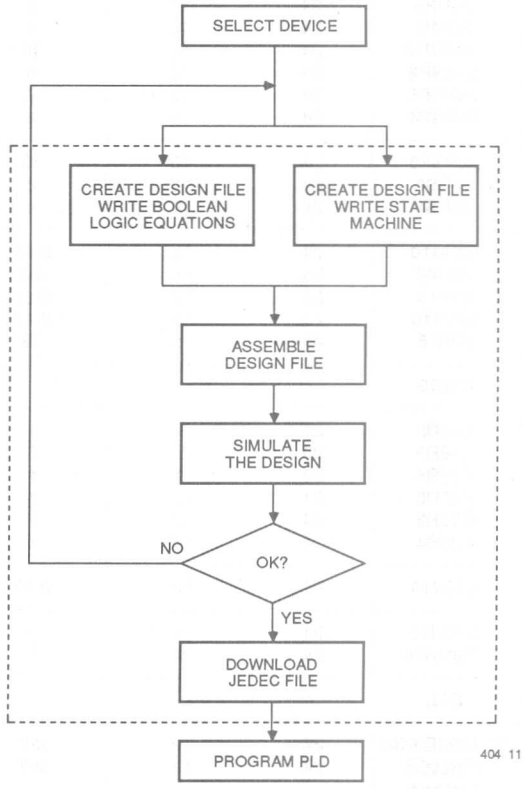


Figure 11. Implementing a Design

```
Title     M68000_ADDRESS_DECODER
Pattern   P8000
Revision  A
Author    ENGINEER
Company   MMI SUNNYVALE, CALIFORNIA
DATE      07/21/87


CHIP DECODER PAL16L8

NC  A23     A22     A21   TOP  BOTTOM  AS  RW  NC  GND
OE  ROMCS1  ROMCS2  RAMCS INIT NC      NC  NC  NC  VCC
```

Figure 12. Declaration Section of a Design File

Design Syntax

As shown in Figure 11, there are two options available to the designer for expressing the design. The first is through traditional Boolean logic equations; the second is through a state machine syntax. The Boolean logic equations are the only option for combinatorial designs and can also be efficient for some registered designs. The Boolean equations can be derived from a combination of the functional description, the truth table and/or the timing diagrams (Figure 13). The state machine approach is ideal for large registered control designs, and can be derived from the functional description, state table, state diagram and/or the timing diagram (Figure 14). The latter approach can also be used with PROSE and PLS devices, and will be discussed in a later section on state machine design.

Boolean Logic Equations

Boolean equations are used to represent the sum-of-products logic form (see page 6-1 for a logic design review). The Boolean equations are ideally suited for representing the two-level AND-OR logic available in most PLDs.

A conventional approach to the design is to convert the design problem to its discrete logic implementation. Such random SSI and MSI logic can be easily implemented in PLDs (see the Beginner's Guide, page 2-2). This usually involves converting to sum-of-products Boolean logic form. This approach can be a chore, and much effort can be saved by implementing a design with PLDs in a sum-of-products form right from the start. This essentially means that the designer does not have to design around the limitations of fixed SSI and MSI functions. A direct implementation of a design in sum-of-products form in a PLD can also yield a faster circuit.

Boolean equations can be directly derived from the truth table or timing diagram (Figure 13). The truth table is used more often in simple combinatorial designs. The timing diagram method is used more often in registered control designs. We will first discuss the truth table method and then discuss the details of the timing diagram method. The state-machine-based registered design approach will be discussed on page 2-101.

In addition to specifying the logic function, the Boolean equations in the design file help document the design. There is no need to draw out an equivalent schematic. This allows design modularity; the schematic can just show a block for a particular PLD. Separate supporting documentation (the design file) provides the details without cluttering the drawing.

Truth-Table-Based Design

The requirements for our particular design example can be easily converted to a truth table format (Figure 15). This truth table is based upon the functional description of the design, and is derived from the address map (Figure 3) and the truth table (Figure 5).

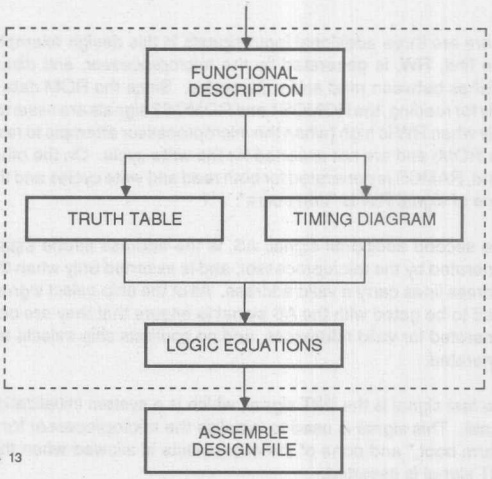
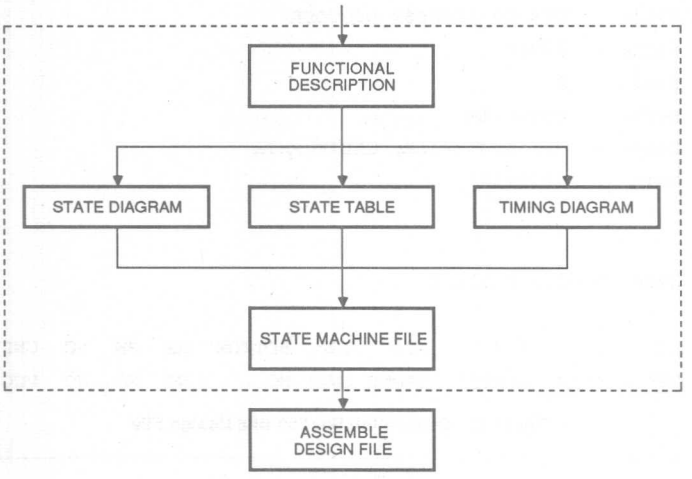


Figure 13. Writing Boolean Logic Equations



404 14

Figure 14. State Machine Dsecription

A23	A22	A21	INIT	AS	RW	OUTPUT GENERATED		
						ROMCS1	ROMCS2	RAMCS
0	0	0	1	0	1	0	1	1
0	0	1	1	0	1	1	0	1
0	1	0	1	0	X	1	1	0
0	1	1	1	0	X	1	1	0
1	0	0	1	0	X	1	1	0
1	0	1	1	0	X	1	1	0

Figure 15. Truth Table for the Address Decoder

There are three additional input signals in this design example. The first, RW, is generated by the microprocessor, and distinguishes between read and write cycles. Since the ROM data is only for reading, the ROMCS1 and ROMCS2 signals are asserted only when RW is high (when the microprocessor attempts to read the ROM) and are not asserted for the write cycle. On the other hand, RAMCS is generated for both read and write cycles and the state of signal RW is "don't care".

The second additional signal, AS, is the address strobe signal generated by the microprocessor, and is asserted only when the address lines carry a valid address. All of the chip select signals need to be gated with the AS signal to ensure that they are only generated for valid addresses, and no spurious chip selects are generated.

The last signal is the INIT signal, which is a system initialization signal. This signal is used to initialize the microprocessor for a "warm boot," and none of the chip selects is allowed when this INIT signal is asserted.

Writing Boolean equations from the above logic is very straightforward. The output signal names, along with their polarity, are assigned to sum-of-product equations, which are based upon inputs and their polarities.

```
/ROMCS1 = /A23 * /A22 * /A21 * INIT * /AS * RW
/ROMCS2 = /A23 * /A22 *  A21 * INIT * /AS * RW
/RAMCS  = /A23 *  A22 * /A21 * INIT * /AS
        + /A23 *  A22 *  A21 * INIT * /AS
        +  A23 * /A22 * /A21 * INIT * /AS
        +  A23 * /A22 *  A21 * INIT * /AS
```

Figure 16. The Implementation in Boolean Equations

The equations are derived directly from the truth tables. Each one of the AND equations uses up one product term of the device as shown in Figure 16. One device selection consideration is to ensure that all the outputs have sufficient product terms to accommodate the desired function. The device we have selected for this design is the popular PAL16L8D. This device has sufficient inputs, outputs, and product terms, and has the requisite speed for the design.

This brings us to the issue of output polarity. Suppose we had to generate active-HIGH outputs. In that case the output equations for the ROMCS1 signal would be :

```
ROMCS1 = /A23 * /A22 * /A21 * INIT * /AS * RW
```

As the PAL16L8 has active-LOW outputs only, this equation's output polarity needs to be inverted to be able to fit the device. Using DeMorgan's theorem for Boolean logic we get:

```
/ROMCS1 = A23 + A22 + A21 + /INIT + AS + /RW
```

This equation requires a large number of product terms (six). Some signals are efficient and use fewer product terms in their true form, while others are more efficient in their inverted form. This poses no problems until the signals exceed the device limit of 8 product terms provided by the PAL16L8 and other standard PAL devices. In such cases, an output polarity control is required. The PAL16RP8 family provides just the control for fitting the required number of product terms on the device. The device selection issues of product terms and output polarity also apply to registered designs. A detailed discussion of output polarity is included on page 6-15.

Timing-Diagram-Based Design

Until now, we have discussed a PLD design using truth tables as the primary design vehicle. In this section we will attempt a design using a timing diagram as a design vehicle.

Earlier in the address decoder design we mentioned the INIT signal. This INIT signal is essentially an initialization signal for the entire system, and can be generated by the PAL16L8. Note that the PAL16L8D used for decoding still has several unused outputs left, one of which can be used to generate the INIT signal. The INIT signal is used internally (via feedback) for disabling the chip selects during initialization. Externally it can be used to initialize other system signals. This INIT signal is generated from a RESET switch connected to the inputs of the PAL16L8 device as shown in Figure 17.

To avoid unwanted initialization, the RESET switch must be debounced. That is, we want the INIT signal to remain HIGH until the switch actually contacts the bottom side. Once the bottom side is hit, INIT should be asserted active LOW. Once asserted, it should stay LOW and not change until the top side is hit again. The timing requirements of the debounce circuitry are shown in Figure 18. Signals TOP and BOTTOM are inputs to the programmable logic device. These signals are activated when the RESET switch touches the top and the bottom contacts, respectively.

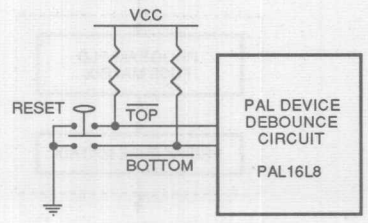


Figure 17. RESET Switch for System Initialization

We can formulate the equations by looking at the timing requirements of the debounce circuitry shown in Figure 18. The idea is to identify the key elements of this timing diagram. The arrows in Figure 18 show the critical events. The first arrow shows the normal state of all the pins when the RESET switch is not asserted. Subsequent arrows show each event in the timing of the INIT signal, depending upon the movement of the switch.

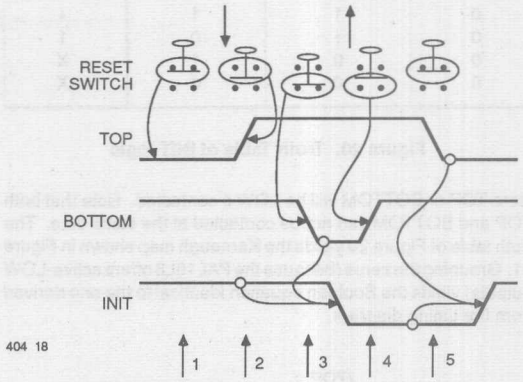


Figure 18. Timing Diagram for the Debounce Switch

The logic level of the signals at each critical event carries useful logic information for deriving Boolean equations. This logic information for each event is converted into direct Boolean equations as shown in Figure 19 below. For example, at instant 1 the INIT signal remains HIGH as long as the TOP signal remains LOW; this is converted to INIT = /TOP * BOTTOM.

1. Normal state — `INIT = /TOP`
2. Switch travels from TOP to BOTTOM — `INIT = TOP * BOTTOM * INIT`
3. Switch contacts BOTTOM — `/INIT = /BOTTOM`
4. Switch travels form BOTTOM to TOP — `/INIT = /INIT * BOTTOM * TOP`
5. Normal State Again

Figure 19. Boolean Logic at Every Instant of Timing

Since we are working with an active-LOW device, we can combine the two active-LOW events into one equation:

```
/INIT = /BOTTOM
      + /INIT * BOTTOM * TOP
```

Minimizing, this becomes:

```
/INIT = /BOTTOM
      + /INIT * TOP
```

This can also be done by way of a truth table and Karnaugh map.

TOP	BOTTOM	INIT–	INIT+
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	X
0	0	0	X

Figure 20. Truth Table of INIT logic

Here TOP or BOTTOM will be LOW if contacted. Note that both TOP and BOTTOM can not be contacted at the same time. The truth table of Figure 20 yields the Karnaugh map shown in Figure 21. Grouping the zeros (because the PAL16L8 offers active-LOW outputs) yields the Boolean equation identical to the one derived from the timing diagram.

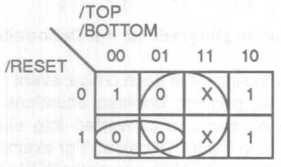


Figure 21. Karnaugh Map of INIT Signal Logic

There is essentially no difference between the truth table and timing diagram techniques for writing Boolean logic. A careful analysis will indicate that we implicitly assumed a truth table in the timing diagram example also. Some designers prefer to make a separate truth table (at least in the first few PLD designs), while others prefer to design directly from timing diagrams. While the truth table method allows a more optimal utilization of product terms, the timing diagram method is easier to visualize as it retains the design perspective. In both cases the logic should be minimized by the design software to ensure that the design is testable. Refer to page 3-108 for a discussion on testability.

Most experienced designers understand the tradeoffs for device selection. They implicitly go through the steps of design conceptualization and device selection, explained earlier. They typically draw a block around the logic being designed, with the previous knowledge that it would fit a PLD which has sufficient inputs, outputs, I/Os and product terms. Now that we have selected a device and implemented the design, the next step is design simulation.

Simulation

Design simulation is an integral part of the design process, as shown in Figure 22. The purpose is to exercise all of the inputs and test the response of outputs to verify that they will work as desired in the system. These are essentially test vectors which designate the state of every input on the device; the outputs are then checked for an appropriate response. The simulation test vectors identify any flaws in the design equations which could affect the logical operation of the devices programmed. Thus the simulation vectors serve as a design debugging tool.

Simulation test vectors will eventually make up part of a larger set of test vectors called "functional test vectors". These functional test vectors are used to exercise a real device after programming to identify any individual devices which are defective. Other means of identifying defective devices, such as signature analysis, are also available. All of these are discussed in detail on page 3-128. In this section we will strictly focus on simulation vectors.

Simulation is included in the design file along with the logic equations. There is little standardization in these simulation expressions among various PLD design software packages, although most of them rely on test vectors to exercise the logic. PALASM 2 software offers an advanced event-driven capability that retains a closer design perspective than simulation vectors. Details of PALASM 2 simulation syntax are included on page 4-137.

The simulation vectors or events can be directly derived from the truth table and the timing diagram of the design. The logic level and functions of all signals can be expanded and rewritten in a test vector form by the software. For example, the truth table for the address decoder example discussed earlier can be easily rewritten as shown in Figure 23.

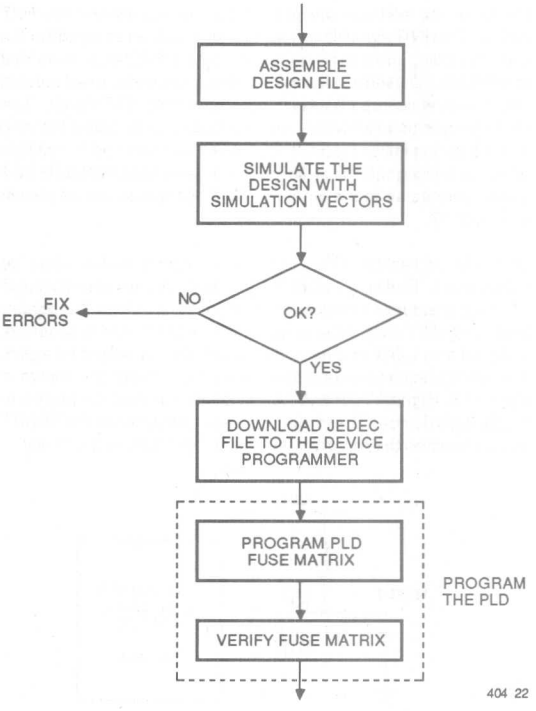


Figure 22. Device Simulation and Programming

A23	A22	A21	TOP	BOTTOM	AS	RW	ROMCS1	ROMCS2	RAMCS	INT
0	0	0	0	1	1	1	H	H	H	H
0	0	0	0	1	0	1	L	H	H	H
0	0	1	0	1	1	1	H	H	H	H
0	0	1	0	1	0	1	H	L	H	H
0	1	0	0	1	1	X	H	H	H	H
0	1	0	0	1	0	X	H	H	L	H
0	1	1	0	1	1	X	H	H	H	H
0	1	1	0	1	0	X	H	H	L	H
1	0	0	0	1	1	X	H	H	H	H
1	0	0	0	1	0	X	H	H	L	H
1	0	1	0	1	1	X	H	H	H	H
1	0	1	0	1	0	X	H	H	L	H
1	0	1	0	1	X	X	H	H	H	H
1	0	1	1	1	X	X	H	H	H	H
1	0	1	1	0	1	X	H	H	H	L
1	0	1	1	1	1	X	H	H	H	L
1	0	1	0	1	1	X	H	H	H	H

Figure 23. Truth Table Used to Derive Simulation Vectors

These are essentially the simulation vectors which will allow us to define the inputs to the device and check the outputs of the device. The PALASM 2 syntax for these simulation vectors is shown in Figure 24.

There is a direct relationship between the truth table vectors of Figure 23 and the simulation shown in Figure 24. The simulator then interprets the design file and generates the output logic levels and/or waveforms, which can be checked by the designer.

Once the simulation is complete, the design file can be assembled to generate the JEDEC file. In the preceding discussions we have assumed prior knowledge of the design file assembly. The procedure for assembly varies with different software packages and is covered for the PALASM 2 software package in the Beginner's Guide (page 2-6).

```
SETF /A23 /A22 /A21 /TOP BOTTOM AS RW
CHECK ROMCS1 ROMCS2 RAMCS INIT

SETF /AS              ; READ CYCLE VALID ROMCS1 ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF AS
SETF /RW /AS          ; WRITE TO ROMCS1 NOT POSSIBLE
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF RW AS

SETF /A23 /A22 A21    ; SET UP ROMCS2 ADDRESS
SETF /AS              ; READ CYCLE VALID ROMCS2 ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF AS
SETF /RW /AS          ; WRITE TO ROMCS2 NOT POSSIBLE
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF RW AS

SETF /A23 A22 /A21    ; SET UP RAMCS ADDRESS
SETF /AS              ; READ CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF AS
SETF /RW /AS          ; WRITE CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF RW AS

SETF /A23 A22 A21     ; SET UP RAMCS ADDRESS
SETF /AS              ; READ CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF AS
SETF /RW /AS          ; WRITE CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF RW AS

SETF A23 /A22 /A21    ; SET UP RAMCS ADDRESS
SETF /AS              ; READ CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF AS
SETF /RW /AS          ; WRITE CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF RW AS

SETF A23 /A22 A21     ; SET UP RAMCS ADDRESS
SETF /AS              ; READ CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF AS
SETF /RW /AS          ; WRITE CYCLE VALID RAMCS ADDRESS
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF RW AS

SETF TOP              ; RESET SWITCH TRAVELING DOWN
SETF /BOTTOM          ; RESET HITTING BOTTOM
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF BOTTOM           ; RESET TRAVELING TO TOP
CHECK ROMCS1 ROMCS2 RAMCS INIT
SETF TOP              ; RESET HITTING TOP
CHECK ROMCS1 ROMCS2 RAMCS INIT
```

Figure 24. PALASM 2 Software Simulation for Address Decoder

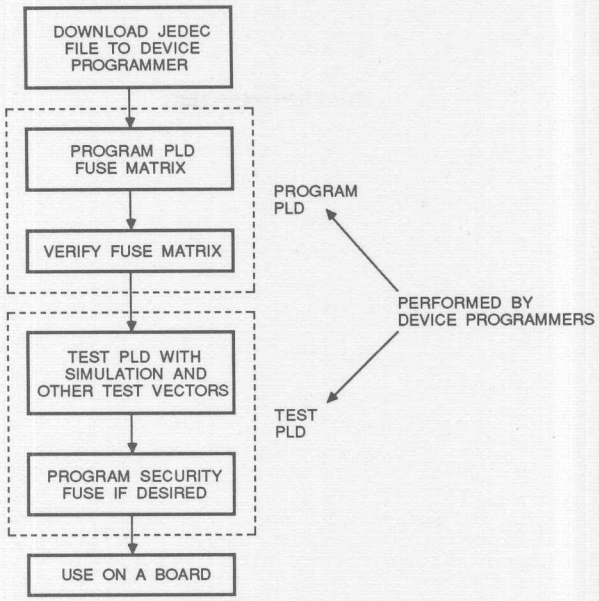


Figure 25. Device Programming and Testing

Device Programming and Testing

Once the design simulation is completed, the final step is device programming and testing (Figure 25). Programmers are available from a variety of vendors. It is important to note that Monolithic Memories qualifies programmers upon verifying that the algorithms used by the programmers are correct and that other basic criteria are met. When purchasing a programmer, check that the programmer is qualified for the devices you intend to use.

There are two types of programmer available; menu-driven or device code based. The menu-driven programmer directly indicates the part type being programmed, whereas the latter type requires the user to enter the device code before programming.

Once the JEDEC fuse file has been downloaded, the programmer can program the device; the PLD is then ready for use. The programmer also verifies the connections after the programming cycle. Programmers also provide the capability of reading a previously programmed device and creating duplicates of that device.

Testing PLDs

The testing of PLDs can be performed by the device programmer or by other test equipment. For a manufacturing environment where high yields are required, device testing is critical. This subject merits a separate discussion and has been included on page 3-128. After testing is complete, the device security cells may be programmed if desired to secure the design from copying.

IMPORTANT
NOTES

Combinatorial Logic Design

In this section we will take a detailed look at several aspects of combinatorial logic design. Most combinatorial design applications can be easily segmented into five major fields.

- Encoders and Decoders
- Multiplexers
- Comparators
- Adders and Arithmetic Logic
- Latches

We will not only focus on the design methodology for these functions, but will also explore further function-specific PLD selection requirements. Generalized designs will be developed, which can be customized later to suit specific system applications. Ways of optimizing the design will also be discussed.

Encoders and Decoders

Two of the most important functions required in digital design are encoding and decoding. The encoding and decoding of data are used extensively in digital communications as well as in peripherals. Both these areas use various complex encoding and decoding techniques, which are described in more detail on pages 2-357 and 2-507. Most of these techniques are extensions of the simple encoding and decoding techniques often used in other digital designs. In this discussion we will focus on simple encoding and decoding techniques.

Encoders

A binary code of n bits can be used to represent 2^n distinct pieces of coded data. A simple combinatorial encoder is a circuit which generates n bits of output information based upon one of the 2^n unique pieces of input data information. This encoding of information is controlled by other independent control signals in a typical digital circuit.

An illustration of a typical encoder is shown in Figure 1. The design methodology typically followed is based on truth tables (Figure 2), from which the Boolean equations are directly derived for the design. The same generic device selection considerations discussed in the section on PAL device design methodology apply for encoder and decoder designs.

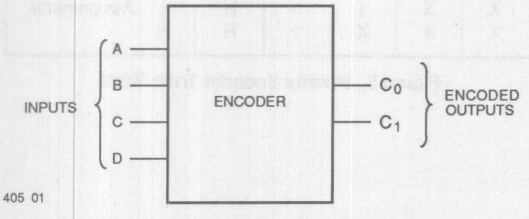


Figure 1. A Block Diagram of an Encoder

INPUTS				OUTPUTS	
A	B	C	D	C_0	C_1
1	0	0	0	L	L
0	1	0	0	L	H
0	0	1	0	H	L
0	0	0	1	H	H

Figure 2. Truth Table of a Typical Encoder

The Boolean equations can then be optimized using Karnaugh maps or the PALASM 2 software minimizer.

The resulting Boolean equations are:

```
C1 = /A *  B * /C * /D   C0 = /A * /B *  C * /D
   + /A * /B * /C *  D       + /A * /B * /C *  D
```

The complete PALASM 2 design file (including the simulation) for this simple encoder design is shown in Figure 3.

A Priority Encoder

Let us take another look at the encoder example of Figure 2. In this example it is assumed that only one of the inputs A, B, C or D is asserted HIGH at any one time. If two of the inputs are asserted HIGH simultaneously, a conflict would be created. To resolve this, a priority needs to be assigned to each of the inputs. Such a priority assignment is used to select a particular element when several inputs are asserted simultaneously. Each input is assigned a priority with respect to the other inputs. The output code generated is the code assigned to the highest priority input asserted.

Thus, a priority encoder is a combinatorial circuit block similar to a general encoder, except that the inputs are assigned a priority. Such priority encoders are used often in state machine applications, where they detect the occurrence of the highest priority event. They are also used for microprocessor interrupt controllers, where they detect the highest priority interrupt. Another use for priority encoders is in bus control, where they are used in arbitration schemes for allowing selective access to the bus.

The model of a priority encoder is shown in Figure 4. The four input signals are A, B, C and D. These are to be encoded as LL, LH, HL and HH outputs. Let us assign priority to D over C, C over B, and B over A. The next design step would be to modify the truth table (Figure 5) to reflect these priorities.

```
TITLE           SIMPLE_ENCODER
PATTERN         ENCODE.PDS
REVISION        01
AUTHOR          JOE ENGINEER
COMPANY         MONOLITHIC MEMORIES, INC.
DATE            8/20/87

; This is an example of a simple encoder. The output C1 and C0
; are generated based upon inputs A, B, C and D. The function
; truth table is following:
;
;       A       B       C       D       C0      C1
;
;       1       0       0       0       L       L
;       0       1       0       0       L       H
;       0       0       1       0       H       L
;       0       0       0       1       H       H
;
;
CHIP ENCODER PAL16H2

A  B  C  D  NC NC NC NC NC GND
NC NC NC NC C0 C1 NC NC NC VCC


EQUATIONS

C1 = /A *  B * /C * /D
   + /A * /B * /C *  D

C0 = /A * /B *  C * /D
   + /A * /B * /C *  D

SIMULATION

TRACE_ON
SETF    A /B /C /D
CHECK /C0 /C1
SETF   /A  B /C /D
CHECK /C0  C1
SETF   /A /B  C /D
CHECK  C0 /C1
SETF   /A /B /C  D
CHECK  C0  C1
```

Figure 3. PALASM 2 File for a Simple Encoder

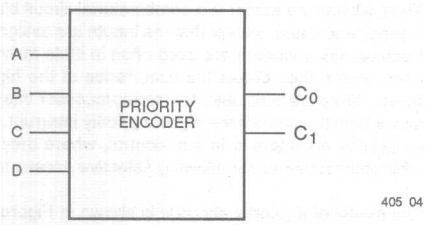


Figure 4. A Four-Input Priority Encoder Block Diagram

INPUTS				OUTPUTS	
A	B	C	D	C_0	C_1
1	0	0	0	L	L
0	1	0	0	L	H
0	0	1	0	H	L
0	0	0	1	H	H
X	1	0	0	L	H Priority
X	X	1	0	H	L Assignments
X	X	X	1	H	H

Figure 5. Priority Encoder Truth Table

The Boolean equations, directly derived from the truth table, are:

```
C1 =  /A *  B * /C * /D   C0  =  /A * /B *  C * /D
    + /A * /B * /C *  D       +  /A * /B * /C *  D
    +      B * /C * /D        +             C * /D
    +              D          +                 D
```

These equations can be further optimized by the design software to the following:

```
C1 = D + /C * B
C0 = D + C
```

Although a priority encoder is a purely combinatorial function, output registers are frequently used to hold the output signal stable for longer durations. An example of a registered 16-input priority encoder is shown in Figure 6.

```
TITLE           16 INPUT REGISTERED PRIORITY ENCODER
PATTERN         P7090
REVISION        01
AUTHOR          J. ENGINEER
COMPANY         MONOLITHIC MEMORIES, INC.
DATE            8/20/87

CHIP IN_PRTY PAL20R4
;
;DESCRIPTION
;
;THE 16 INPUT REGISTERED PRIORITY ENCODER ACCEPTS SIXTEEN ACTIVE-LOW INPUTS
;(I0-I15) TO LOAD THE BINARY WEIGHTED CODE OF THE PRIORITY ORDER INTO THE
;OUTPUT REGISTER (Q3-Q0) ON THE RISING EDGE OF THE CLOCK (CLK).  A PRIORITY
;IS ASSIGNED TO EACH INPUT SO THAT WHEN TWO INPUTS ARE SIMULTANEOUSLY
;ACTIVE, THE INPUT WITH THE HIGHEST PRIORITY IS LOADED INTO THE OUTPUT
;REGISTER.  THEREFORE THE HIGHEST PRIORITY INPUT (I0=H) PRODUCES HHHH IN
;THE OUTPUT REGISTER AND THE LOWEST PRIORITY INPUT (I15=H) PRODUCES LLLL IN
;THE OUTPUT REGISTER.
;
;OPERATIONS TABLE
;
;       /OC  CLK  I15-I0   Q3-Q0    OPERATION
;       ------------------------------------------------------------------
;        H    X      X       Z      HI-Z
;        L    C    I0 =H    15      I0  INTERRUPT (HIGHEST PRIORITY INPUT)
;        L    C    I1 =H    14      I1  INTERRUPT
;        L    C    I1 =H    13      I2  INTERRUPT
;        L    C    I1 =H    12      I3  INTERRUPT
;        L    C    I1 =H    11      I4  INTERRUPT
;        L    C    I1 =H    10      I5  INTERRUPT
;        L    C    I1 =H     9      I6  INTERRUPT
;        L    C    I1 =H     8      I7  INTERRUPT
;        L    C    I1 =H     7      I8  INTERRUPT
;        L    C    I1 =H     6      I9  INTERRUPT
;        L    C    I1 =H     5      I10 INTERRUPT
;        L    C    I1 =H     4      I11 INTERRUPT
;        L    C    I1 =H     3      I12 INTERRUPT
;        L    C    I1 =H     2      I13 INTERRUPT
;        L    C    I1 =H     1      I14 INTERRUPT
;        L    C    I1 =H     0      I15 INTERRUPT (LOWEST PRIORITY INPUT)
;       ------------------------------------------------------------------
;
;

CLK I0  I1  I2  I3 I4 I5 I6 I7  I8  I9  GND
/OC I10 I11 I12 Q3 Q2 Q1 Q0 I13 I14 I15 VCC

EQUATIONS


/Q0 := /I0* I1
     + /I0*/I1*/I2* I3
     + /I0*/I1*/I2*/I3*/I4* I5
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6* I7
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8* I9
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10* I11
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12* I13
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12*/I13*/I14*I15
```

Figure 6. A 16-Input Registered Priority Encoder Design File

```
/Q1 := /I0*/I1* I2
     + /I0*/I1*/I2* I3
     + /I0*/I1*/I2*/I3*/I4*/I5* I6
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6* I7
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9* I10
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10* I11
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12*/I13* I14
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12*/I13*/I14*I15

/Q2 := /I0*/I1*/I2*/I3* I4
     + /I0*/I1*/I2*/I3*/I4* I5
     + /I0*/I1*/I2*/I3*/I4*/I5* I6
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6* I7
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11* I12
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12* I13
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12*/I13* I14
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12*/I13*/I14*I15

/Q3 := /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7* I8
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8* I9
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9* I10
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10* I11
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11* I12
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12* I13
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12*/I13* I14
     + /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7*/I8*/I9*/I10*/I11*/I12*/I13*/I14*I15

; SIMULATION NOT INCLUDED HERE
```

Figure 6. A 16-Input Registered Priority Encoder Design File (Cont'd.)

INPUT SELECT LINES				OUTPUT LINES														
A	B	C	D	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15
0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Figure 7. The Truth Table of an Active-LOW 4-to-16 Decoder

Decoders

A decoder performs the reverse function of an encoder. It converts an n-bit code to one of its 2^n unique items. It is a combinatorial circuit designed such that at most one of its several outputs will be asserted based upon the unique input codes.

A decoder may have as many outputs as there are possible binary input selection combinations. As shown in the truth table (Figure 7), only one output may be asserted at any time. When a new combination is applied, another output is asserted and the original output is returned to its non-asserted state.

The Boolean logic equations can be directly derived from the truth table shown in Figure 7. The procedure is the same as explained in the previous section on PLD design methodology on page 2-27. The Boolean equations derived are shown in Figure 8.

```
/Q0  = /D * /C * /B * /A
/Q1  = /D * /C * /B *  A
/Q2  = /D * /C *  B * /A
/Q3  = /D * /C *  B *  A
/Q4  = /D *  C * /B * /A
/Q5  = /D *  C * /B *  A
/Q6  = /D *  C *  B * /A
/Q7  = /D *  C *  B *  A
/Q8  =  D * /C * /B * /A
/Q9  =  D * /C * /B *  A
/Q10 =  D * /C *  B * /A
/Q11 =  D * /C *  B *  A
/Q12 =  D *  C * /B * /A
/Q13 =  D *  C * /B *  A
/Q14 =  D *  C *  B * /A
/Q15 =  D *  C *  B *  A
```

Figure 8. Decoder Boolean Logic Equations

Notice from the truth table that there is no combination of inputs that will send all the outputs to their non-asserted state. Many designs actually need to be able to make all outputs inactive. This can be done simply by putting enable lines in all of the output AND gates. Many such design modifications can be easily added once the basic Boolean equations have been derived, instead of redoing the truth table. Figure 9 shows the PALASM 2 software design file of one such 4-to-16 decoder with the enable function.

Probably the most commonly used decoders are the address decoders required by most microprocessors and bus interfaces. These also constitute the most common application of PLDs in digital designs. The design considerations for address decoders have been covered earlier in the PLD Design Methodology section, page 2-23. Later we will develop a general Boolean equation for an address decoder circuit when we discuss range decoders.

```
Title      4to16 Decoder
Pattern    4-16DEC.PDS
Revision   A
Author     Mehrnaz Hada
Company    Monolithic Memories, Santa Clara, CA
Date       1/9/85

CHIP Decoder PAL6L16

Q0 Q1 Q2 A  B  C   D   EN1 EN2 Q3  Q4  GND
Q5 Q6 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15 VCC

;The 4 to 16 decoder decodes four binary encoded inputs
;into one of 16 mutually exclusive outputs, whenever the
;two enable lines EN1 and EN2 are high. When one or both
;of the enable lines are low the outputs are all set to
;high values.

EQUATIONS

/Q0  = /D*/C*/B*/A* EN1* EN2          ;Decode 0000
/Q1  = /D*/C*/B* A* EN1* EN2          ;Decode 0001
/Q2  = /D*/C* B*/A* EN1* EN2          ;Decode 0010
/Q3  = /D*/C* B* A* EN1* EN2          ;Decode 0011
/Q4  = /D* C*/B*/A* EN1* EN2          ;Decode 0100
/Q5  = /D* C*/B* A* EN1* EN2          ;Decode 0101
/Q6  = /D* C* B*/A* EN1* EN2          ;Decode 0110
/Q7  = /D* C* B* A* EN1* EN2          ;Decode 0111
/Q8  =  D*/C*/B*/A* EN1* EN2          ;Decode 1000
/Q9  =  D*/C*/B* A* EN1* EN2          ;Decode 1001
/Q10 =  D*/C* B*/A* EN1* EN2          ;Decode 1010
/Q11 =  D*/C* B* A* EN1* EN2          ;Decode 1011
/Q12 =  D* C*/B*/A* EN1* EN2          ;Decode 1100
/Q13 =  D* C*/B* A* EN1* EN2          ;Decode 1101
/Q14 =  D* C* B*/A* EN1* EN2          ;Decode 1110
/Q15 =  D* C* B* A* EN1* EN2          ;Decode 1111


SIMULATION

TRACE_ON  D B C A Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10
          Q11 Q12 Q13 Q14 Q15

SETF /D /C /B /A EN1 EN2
SETF A
SETF B
SETF C
SETF D
SETF /D
SETF /C
SETF /B
SETF /A
SETF /EN1                             ;Set outputs to high
SETF EN1 /EN2                         ;Set outputs to high
SETF /EN1                             ;Set outputs to high

TRACE_OFF
```

Figure 9. Design File of Four-to-Sixteen Decoder with Enable

Encoder/Decoder Device Selection Considerations

The general device selection considerations are listed in Figure 10. Based upon the number of inputs and outputs required, a device can be selected from the table in Figure 11 for combinatorial devices or Figure 12 for registered devices.

- Number of Input Pins
- Number of Output Pins
- Number of I/O Pins
- Device Speed
- Device Power Requirements
- Number of Registers
- Number of Product Terms
- Output Polarity Control

Figure 10. General Encoder/Decoder Device Selection Considerations

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES t_{PD} IN ns
TTL						
10H8	20	10	8	-	2	35
10L8	20	10	8	-	2	35
12H6	20	12	6	-	2-4	35
12L6	20	12	6	-	2-4	35
14H4	20	14	4	-	4	35
14L4	20	14	4	-	4	35
16H2	20	16	2	-	8	35
16L2	20	16	2	-	8	35
16C1	20	16	2	-	16	40
16L8	20	10	2	6	7	10, 15, 25, 35
16P8	20	10	2	6	7	25
16RA8	20	8	-	8	4	30
18P8	20	10	-	8	8	15, 25
6L16	24	6	16	-	1	25
8L14	24	8	14	-	1	25
12L10	24	12	10	-	2	40
14L8	24	14	8	-	2-4	40
16L6	24	16	6	-	2-4	40
18L4	24	18	4	-	4-6	40
20L2	24	20	2	-	8	40
20C1	24	20	2	-	16	40
20L8	24	14	2	6	7	15, 25, 35
20L10	24	12	2	8	3	15, 20, 25, 30
20RA10	24	10	-	10	4	20, 30
20S10	24	12	2	8	8-16	35
22P10	24	12	-	10	8	15, 25
22XP10	24	12	-	10	8	20, 30, 40
22RX8	24	14	-	8	8	25
22V10	24	12	-	10	8-16	15, 25, 35
32VX10	24	12	-	10	8-16	25, 30
CMOS						
C16L8	20	10	2	6	7	25
C20L8	24	14	2	6	7	35, 45
C22V10	24	12	-	10	8-16	25, 35
C29M16	24	5	-	16	8-16	35, 45
C29MA16	24	5	-	16	4-12	35, 45
ECL						
10H20P8	24	12	-	8	4-8	6

Figure 11. Combinatorial Programmable Logic Devices

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16R8	20	8	8	-	8	55.5, 37, 25, 16
16R6	20	8	6	2	8	55.5, 37, 25, 16
16R4	20	8	4	4	8	55.5, 37, 25, 16
16RP8	20	8	8	-	8	22.2
16RP6	20	8	6	2	8	22.2
16RP4	20	8	4	4	8	22.2
16RA8	20	8	0-8	8-0	4	20
16X4	20	8	4	4	8	14
23S8	20	9	4	4	8-12	33.3, 28.5
20R8	24	12	8	-	8	37, 25, 16
20R6	24	12	6	2	8	37, 25, 16
20R4	24	12	4	4	8	37, 25, 16
20X10	24	10	10	-	4	22.2
20X8	24	10	8	2	4	22.2
20X4	24	10	4	6	4	22.2
20RP10	24	10	10	-	8	25, 37
20RP8	24	10	8	2	8	25, 37
20RP6	24	10	6	4	8	25, 37
20RP4	24	10	4	6	8	25, 37
20XRP10	24	10	10	-	8	30, 22.2, 14
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
20RS10	24	10	10	-	8-16	19.2
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
32R16	40	16	16	-	8-16	16
CMOS						
C16R8	20	8	8	-	8	28.5
C16R6	20	8	6	2	8	28.5
C16R4	20	8	4	4	8	28.5
C20R8	24	12	8	-	8	20, 15.3
C20R6	24	12	6	2	8	20, 15.3
C20R4	24	12	4	4	8	20, 15.3
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
10H/020EV/G8	24	12	0-8	8-0	8-12	125
10H20G8 (Latched)	24	12	0-8	8-0	4-8	NA

Figure 12. Registered Programmable Logic Devices

Encoders typically require a large number of inputs and fewer outputs, whereas decoders typically require a large number of outputs and fewer inputs. We can see from these tables that the PAL device for the priority encoder application (Figure 6) should be the PAL20R4 and for the decoder application (Figure 7) it should be the PAL6L16.

Another important device selection consideration for encoders and decoders is the number of product terms required for a design. A careful selection of code values (and priority assignments in priority encoders) can often reduce the required number of product terms. This can sometimes determine whether or not a design fits a device successfully. Figure 13 shows the truth tables of two simple partial 3-to-2 encoders. The product terms required for the two designs are different due to the different assignment of encoded bits.

INPUTS			OUTPUTS	
A	B	C	X_1	X_0
1	0	0	0	0
0	1	0	0	1
0	0	1	1	0

```
X1 = /A * /B *  C

X0 = /A *  B * /C
```

INPUTS			OUTPUTS	
A	B	C	X_1	X_0
1	0	0	0	1
0	1	0	1	0
0	0	1	1	1

```
X1 = /A *  B * /C
   + /A * /B *  C

X0 =  A * /B * /C
   + /A * /B *  C
```

Figure 13. Two Encoders With Different Product Term Requirements

Another way of looking at a decoder is as a logic function which, depending upon the select code applied, connects one data input to the selected outputs. Also known as a Demultiplexer, a decoder essentially connects an input to one of 2^n outputs based upon n select code bits. The reverse logic function, which combines data from multiple sources to an output signal, is called a multiplexer and is discussed next.

Multiplexers

A multiplexer (sometimes referred to as a data selector) is a special combinatorial circuit, widely used in digital design. It is designed to gate one of several inputs to a single output. The input selected for connection to the output is controlled by a separate set of select inputs.

The traditional use of a multiplexer is for "time division multiplexing" in data communication, when gating several data lines to a single data transmission line for short intervals of time. The data received is then demultiplexed by using a demultiplexer, described earlier.

The design methodology employed for multiplexer design is the truth-table approach. As an example, we can look at a three-input-to-one-output (3:1) multiplexer, which uses two select signals A and B. Based on these two select bits, the data on one of the three inputs is sent to the output. The truth table is shown in Figure 14.

SELECT		INPUTS			OUTPUT
B	A	I1C0	I1C1	I1C2	O1Y
0	0	0	X	X	0
0	0	1	X	X	1
0	1	X	0	X	0
0	1	X	1	X	1
1	0	X	X	0	0
1	0	X	X	1	1

Figure 14. Truth Table for a Three-to-One Multiplexer

Deriving the Boolean equation from this truth table is a straightforward task. In this case no further minimization is possible. The Boolean equation is:

```
/O1Y = /B * /A * /I1C0
     + /B *  A * /I1C1
     +  B * /A * /I1C2
```

The complete PALASM 2 design file for this design example showing four such three-to-one multiplexers is shown in Figure 15.

```
TITLE           QUAD 3:1 MULTIPLEXER
PATTERN         P7074
REVISION        01
AUTHOR          COLI/VOLPIGNO
COMPANY         MONOLITHIC MEMORIES, INC.
DATE            8/20/87

;DESCRIPTION
;
;THIS IS AN EXAMPLE OF A QUAD 3-TO-1 MULTIPLEXER USING A PAL14L4.  SELECT
;LINES A,B ARE ENCODED IN BINARY, WITH A REPRESENTING THE LSB.  THE OUTPUTS
;(Y) ARE ALL HIGH IF THE SELECT LINES ARE BOTH HIGH (B,A=H).
;
;       OPERATIONS TABLE:
;
;       INPUT    OUTPUTS
;       SELECT
;        B  A       Y
;       ----------------
;        L  L       C0
;        L  H       C1
;        H  L       C2
;        H  H       H
;       ----------------
;
CHIP QUAD_MUX PAL14L4

I1C0 I1C1 I1C2 I2C0 I2C1 I2C2 I3C0 I3C1 I3C2 GND
I4C0 I4C1 I4C2  O4Y  O3Y  O2Y  O1Y   B    A  VCC

EQUATIONS

/O1Y =  /B*/A * /I1C0                          ;SELECT INPUT 1C0
     +  /B* A * /I1C1                          ;SELECT INPUT 1C1
     +   B*/A * /I1C2                          ;SELECT INPUT 1C2

/O2Y =  /B*/A * /I2C0                          ;SELECT INPUT 2C0
     +  /B* A * /I2C1                          ;SELECT INPUT 2C1
     +   B*/A * /I2C2                          ;SELECT INPUT 2C2

/O3Y =  /B*/A * /I3C0                          ;SELECT INPUT 3C0
     +  /B* A * /I3C1                          ;SELECT INPUT 3C1
     +   B*/A * /I3C2                          ;SELECT INPUT 3C2

/O4Y =  /B*/A * /I4C0                          ;SELECT INPUT 4C0
     +  /B* A * /I4C1                          ;SELECT INPUT 4C1
     +   B*/A * /I4C2                          ;SELECT INPUT 4C2


; SIMULATION NOT INCLUDED HERE
```

Figure 15. Quad 3:1 Multiplexer

The equations derived in the above example can be easily generalized for other multiplexers. The symbol for a general 2^n-inputs-to-one-output multiplexer is shown in Figure 16 where n select lines are used.

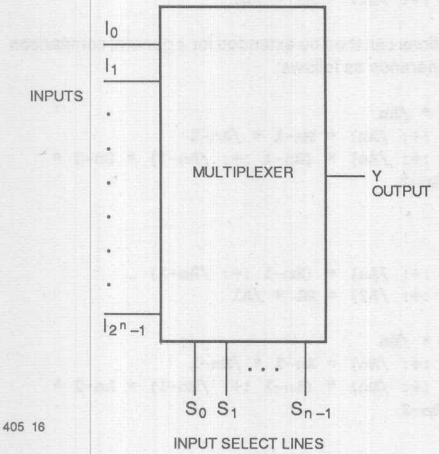


Figure 16. General Model of a 2^n-to-1 Multiplexer

The Boolean equations are:

```
n=2

     Y = /S1*/S0*(I0)
       + /S1* S0*(I1)
       +  S1*/S0*(I2)
       +  S1* S0*(I3)

n=3

     Y = /S2*/S1*/S0*(I0)
       + /S2*/S1* S0*(I1)
       + /S2* S1*/S0*(I2)
       + /S2* S1* S0*(I3)
       +  S2*/S1*/S0*(I4)
       +  S2*/S1* S0*(I5)
       +  S2* S1*/S0*(I6)
       +  S2* S1* S0*(I7)
```

Multiplexer Device Selection Considerations

Multiplexers typically require more inputs than outputs, so the devices with a large number of inputs and I/Os are usually more useful. Careful consideration must also be given to the number of product terms available on each output.

Several multiplexers are often used simultaneously to route multiple address and data bits, under the control of the same select lines. In such cases, multiple devices can be cascaded when the number of inputs and outputs exceeds device limits. Cascading is also possible for large multiplexers that do not fit in a single device. In such cases, the select bits should also be judiciously selected for each PLD, to minimize the number of product terms.

Another common trick for designing a multiplexer is to connect a number of outputs together and control the output enables using the select bits to multiplex data. This multiplexing technique is used extensively in DRAM controllers for multiplexing address signals. Timing considerations for such designs include the output enable and disable times, which should be carefully selected to avoid output contentions.

Comparators

A comparator is a combinatorial circuit designed primarily to compare the relative magnitude of two binary numbers. Figure 17 shows the truth table for a two-bit comparator.

INPUTS				OUTPUTS		
A		B		EQL	LES	GTR
A_2	A_1	B_2	B_1	A=B	A<B	A>B
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

Figure 17. Truth Table for a Comparator

A basic comparator compares two numbers only for equality, and generates the EQL signal (indicating A=B). An extension, called a magnitude comparator, also generates the LES signal (indicating A<B) and GTR signal (indicating A>B). Based on this truth table, the equations for the three output signals EQL, LES and GTR can be easily derived. These equations can then be optimized by using Boolean algebra, Karnaugh maps or the minimization routine available with PALASM 2 software.

The final Boolean equations are:

```
EQL =  /A2 * /A1 * /B2 * /B1
     + /A2 *  A1 * /B2 *  B1
     +  A2 * /A1 *  B2 * /B1
     +  A2 *  A1 *  B2 *  B1

LES =  /A2 * /A1 * /B2 *  B1
     + /A2 * /A1 *  B2 * /B1
     + /A2 * /A1 *  B2 *  B1
     + /A2 *  A1 *  B2 * /B1
     + /A2 *  A1 *  B2 *  B1
     +  A2 * /A1 *  B2 *  B1

     = /A1 *  B2 *  B1
     + /A2 * /A1 *  B1
     + /A2 *  B2

GTR =  /A2 *  A1 * /B2 * /B1
     +  A2 * /A1 * /B2 * /B1
     +  A2 *  A1 * /B2 * /B1
     +  A2 * /A1 * /B2 *  B1
     +  A2 *  A1 * /B2 *  B1
     +  A2 *  A1 *  B2 * /B1

     =  A1 * /B2 * /B1
     +  A2 *  A1 * /B1
     +  A2 * /B2
```

Comparator Device Selection Considerations

The number of product terms needed is directly related to the number of bits compared. For LES (less than) and GTR (greater than) functions, the number of product terms required depends upon the number of bits in the two operands compared, as well as their value. The LES and GTR equations can be rewritten as follows:

```
LES =  B2 * /A2
     + (B2 :+: /A2) * B1 * /A1

GTR =  A2 * /B2
     + (A2 :+: /B2) * A1 * /B1
```

These equations can then be extended for a general comparison of n-bit comparands as follows:

```
LES =  Bn * /An
     + (Bn :+: /An) * Bn-1 * /An-1
     + (Bn :+: /An) * (Bn-1 :+: /An-1) * Bn-2 *
       /An-2
     + ...
     + ...
     + ...
     + (Bn :+: /An) * (Bn-1 :+: /An-1) ...
       (B2 :+: /A2) * B1 * /A1

GTR =  An * /Bn
     + (An :+: /Bn) * An-1 * /Bn-1
     + (An :+: /Bn) * (An-1 :+: /Bn-1) * An-2 *
       /Bn-2
     + ...
     + ...
     + ...
     + (An :+: /Bn) * (An-1 :+: /Bn-1) ...
       (A2 :+: /B2) * A1 * /B1
```

The total number of product terms required for an n-bit comparison is 2^n-1. Comparators require a large number of product terms and devices that offer many product terms can be used very effectively. Figure 18 shows a list of devices that offer many product terms. It also includes registered devices, since registers are sometimes used to hold the comparator output signals asserted for long durations.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES t_{PD} IN ns
TTL						
16C1	20	16	2	-	16	40
20C1	24	20	2	-	16	40
20S10	24	12	2	8	8-16	35
22V10	24	12	-	10	8-16	25, 35
32VX10	24	12	-	10	8-16	25, 35
CMOS						
C22V10	24	12	-	10	8-16	25, 35
C29M16	24	5	-	16	8-16	35, 45
C29MA16	24	5	-	16	4-12	35, 45

Figure 18. PLDs With Large Numbers of Product Terms

As is obvious from these equations, comparators require exclusive-OR functions, and can also be efficiently implemented in the PAL16X4, which offers exclusive-OR functions. An example of a four-bit comparator implemented on a PAL16X4 is shown in Figure 19.

```
PAL16X4                                          PAL DESIGN SPECIFICATION
P7031                                              BIRKNER/COLI 07/12/81
BETWEEN LIMITS COMPARATOR/REGISTER
MMI SUNNYVALE, CALIFORNIA

 CLK LOAD CLEAR B0 B1 B2 B3  NC /OC2 GND
/OC1 /NE   /EQ  A3 A2 A1 A0 /LT /GT  VCC

;DESCRIPTION

;THE DEVICE CONTINUOUSLY COMPARES THE BUS VALUE (B3-B0) WITH THE VALUE OF
;THE REGISTER (A3-A0) AND REPORTS THE STATUS ON OUTPUTS LT, EQ, NE, AND GT:

;       * LT INDICATES THAT B IS  LESS   THAN A
;       * EQ INDICATES THAT B IS   EQUAL   TO A
;       * NE INDICATES THAT B IS NOT EQUAL TO A
;       * GT INDICATES THAT B IS GREATER THAN A

;                                  STATUS       BUS    REG
;/OC1  /OC2  CLK  LOAD  CLEAR  LT EQ NE GT  B3-B0  A3-A0    OPERATION
;-------------------------------------------------------------------------
; H      X    X    X      X    X  X  X  X     X       Z      REG HI-Z
; X      H    X    X      X    Z  Z  Z  Z     X       X      STATUS HI-Z
; L      X    X    L      L    X  X  X  X     X       A      READ REG
; X      X    C    H      L    X  X  X  X     B       B      LOAD REG
; X      X    C    L      L    X  X  X  X     X       A      HOLD
; X      X    C    X      H    X  X  X  X     X       L      CLEAR REG
; X      L    X    L      L      STATUS       B       A      COMPARE
;-------------------------------------------------------------------------

IF(OC2)  LT = (A3*/B3)                                              ;B3=L, A3=H
           + (A3:*:B3) * (A2*/B2)                                   ;B2=L, A2=H
           + (A3:*:B3) * (A2:*:B2) * (A1*/B1)                       ;B1=L, A1=H
           + (A3:*:B3) * (A2:*:B2) * (A1:*:B1) * (A0*/B0)           ;B0=L, A0=H

IF(OC2)  GT = (/A3*B3)                                              ;B3=H, A3=L
           + (A3:*:B3) * (/A2*B2)                                   ;B2=H, A2=L
           + (A3:*:B3) * (A2:*:B2) * (/A1*B1)                       ;B1=H, A1=L
           + (A3:*:B3) * (A2:*:B2) * (A1:*:B1) * (/A0*B0)           ;B0=H, A0=L

IF(OC2)  EQ = (A3:*:B3) * (A2:*:B2) * (A1:*:B1) * (A0:*:B0) ;COMPARE EQUAL

IF(OC2)  NE = (A3:+:B3) + (A2:+:B2) + (A1:+:B1) + (A0:+:B0) ;NOT EQUAL

       /A3 := (/A3)*/LOAD      ;HOLD  REG A3
            + (/B3)* LOAD      ;LOAD  REG A3
            +  CLEAR           ;CLEAR REG A3

       /A2 := (/A2)*/LOAD      ;HOLD  REG A2
            + (/B2)* LOAD      ;LOAD  REG A2
            +  CLEAR           ;CLEAR REG A2

       /A1 := (/A1)*/LOAD      ;HOLD  REG A1
            + (/B1)* LOAD      ;LOAD  REG A1
            +  CLEAR           ;CLEAR REG A1

       /A0 := (/A0)*/LOAD      ;HOLD  REG A0
            + (/B0)* LOAD      ;LOAD  REG A0
            +  CLEAR           ;CLEAR REG A0
```

Figure 19. PAL16X4 Based Four-Bit Comparator

FUNCTION TABLE

CLK /OC1 /OC2 LOAD CLEAR B3 B2 B1 B0 A3 A2 A1 A0 LT EQ NE GT

;CONTROL ; /OC ;CLK 1 2	OPERATIONS LOAD CLEAR	BUS BBBB 3210	REG AAAA 3210	--STATUS--- LT EQ NE GT	COMMENTS (HEX VALUES)
C L X	X H	XXXX	LLLL	X X X X	CLEAR REG
C L X	H L	LLLL	LLLL	X X X X	LOAD REG (0)
X L L	L L	LLLL	LLLL	L H L L	COMPARE (0 EQ 0)
X L L	L L	LLLH	LLLL	L L H H	COMPARE (1 GT 0)
X L X	L L	XXXX	LLLL	X X X X	READ REG (0)
C L X	X H	XXXX	LLLL	X X X X	CLEAR REG
C L X	H L	LHLH	LHLH	X X X X	LOAD REG (5)
X L L	L L	LLLL	LHLH	H L H L	COMPARE (0 LT 5)
X L L	L L	LHLH	LHLH	L H L L	COMPARE (5 EQ 5)
X L L	L L	HHHH	LHLH	L L H H	COMPARE (F GT 5)
X L X	L L	XXXX	LHLH	X X X X	READ REG (5)
C L X	X H	XXXX	LLLL	X X X X	CLEAR REG
C L X	H L	HLHL	HLHL	X X X X	LOAD REG (A)
X L L	L L	LHLL	HLHL	H L H L	COMPARE (4 LT A)
X L L	L L	HLHL	HLHL	L H L L	COMPARE (A EQ A)
X L L	L L	HLHH	HLHL	L L H H	COMPARE (B GT A)
X L X	L L	XXXX	HLHL	X X X X	READ REG (A)
C L X	X H	XXXX	LLLL	X X X X	CLEAR REG
C L X	H L	HHHH	HHHH	X X X X	LOAD REG (F)
X L L	L L	HHHL	HHHH	H L H L	COMPARE (E LT F)
X L L	L L	HHHH	HHHH	L H L L	COMPARE (F EQ F)
X L X	L L	XXXX	HHHH	X X X X	READ REG (F)
C L X	L L	XXXX	HHHH	X X X X	HOLD (F)
X H X	X X	XXXX	ZZZZ	X X X X	TEST HI-Z (/OC1=H)
X X H	X X	XXXX	XXXX	Z Z Z Z	TEST HI-Z (/OC2=H)

Figure 19. PAL16X4 Based Four-Bit Comparator (Cont'd.)

The values of the comparands themselves affect the number of product terms used. When the comparison is made with comparands which are power-of-two numbers, the number of product terms required can be reduced drastically. This essentially relies on the fact that when the lower bits of a comparand are all zeros only the highest bit needs to be compared, requiring only one product term. For example in a two-bit comparator, if A1 is zero and A2 is one, the equation for the greater-than function becomes very simple and requires only one product term:

```
GTR = /B2
```

The general equation for the GTR signal can also be simplified when comparing a number B to a fixed power-of-two comparand A with p least significant zeros.

```
A = 000010000 ... 00
    n   p       1

GTR =  /Bn * /Bn-1 ... * /Bp+1 * /Bp
```

This general GTR equation can also be considered as an equation for comparing a number to a range of numbers extending from zero to number A. In fact this trick is used very often by many system designers for address decoder functions. In PLD Design Methodology (page 2-22) the ROMCS1 signal is one such signal which is generated for the address range from (000000) hex to (0FFFFF) hex. For this design n=23, the comparand A=(0FFFFF + 1)=100000 and p=21. Substituting in the general equation we get the same address decoder Boolean logic equation.

```
ROMCS1 = /A23 * /A22 * /A21
```

As such designs require few product terms and no XOR gates, they are efficiently implemented on standard combinatorial PLDs. A list of such standard combinatorial PLDs is shown in Figure 20. A general form of range comparators with two boundary comparands will be discussed later.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES t_{PD} IN ns
TTL						
16L8	20	10	2	6	7	10, 15, 25, 35
16P8	20	10	2	6	7	25
18P8	20	10	-	8	8	15, 25
20L8	24	14	2	6	7	15, 25, 35
20L10	24	12	2	8	3	15, 20, 25, 30
20S10	24	12	2	8	8-16	35
22P10	24	12	-	10	8	15, 25
22XP10	24	12	-	10	8	20, 30, 40
22RX8	24	14	-	8	8	25
22V10	24	12	-	10	8-16	25, 35
32VX10	24	12	-	10	8-16	25, 30
CMOS						
C16L8	20	10	2	6	7	25
C20L8	24	14	2	6	7	35, 45
C22V10	24	12	-	10	8-16	25, 35
C29M16	24	5	-	16	8-16	35, 45
C29MA16	24	5	-	16	4-12	35, 45

Figure 20. Standard Combinatorial PLDs with Sum-of-Products Logic

The third output signal is the EQL signal. The EQL Boolean equation tells us whether the two numbers are identical. Such information is useful not only in address decoders, but also in digital signal processing designs. This equation requires a large number of product terms. A closer examination reveals that it is essentially an exclusive-OR function.

```
EQL =  /A2 * /B2 * (/A1 * /B1 + A1 * B1)
     + A2 *  B2 * (/A1 * /B1 + A1 * B1)

EQL = (A1 :*: B1) * (A2 :*: B2) ; Exclusive-NOR
                                ; function
```

Inverting this:

```
/EQL = (A1 :+: B1) + (A2 :+: B2) ; Exclusive-OR
                                 ; function
```

This equation can be extended to give a general equation for equal-to comparison for two n-bit comparands.

```
/EQL =  (An :+: Bn)
      + (An-1 :+: Bn-1)
      + (An-2 :+: Bn-2)
      + (An-3 :+: Bn-3)
      + ...
      + ...
      + (A1 :+: B1)
```

This inverted equation is implementable in the sum-of-products form of the exclusive-OR functions, and can be easily expanded to the following:

```
/EQL =  A1 * /B1 + /A1 * B1
      + A2 * /B2 + /A2 * B2
      + A3 * /B3 + /A3 * B3
      + ...
      + ...
      + An * /Bn + /An * Bn
```

This gives us a general sum-of-products form of a comparator equation which is easily implemented in PAL devices. An n-bit comparator requires 2n product terms. An eight-bit comparator is implemented in the design shown in Figure 21. This design uses the PAL16C1, which is specifically designed for comparator applications, and provides a large number of product terms connected to EQL and /EQL (NE) outputs. All of the devices that have a large number of product terms are illustrated in Figure 18, and would be appropriate for such designs.

```
Title     Octal_Comparator
Pattern   OctComp.pds
Revision  A
Author    Mehrnaz Hada
Company   Monolithic Memories Inc., Santa Clara,CA
Date      1/29/85

;The octal comparator establishes when two 8-bit data
;strings (A7-A0) and (B7-B0) are equivalent (EQ=H) or not
;equivalent (NE=H).

CHIP OctalCom PAL16C1

A7 A0 B0 A1 B1 A2 B2 A3 B3 GND
A4 B4 A5 B5 EQ NE A6 B6 B7 VCC

EQUATIONS

NE =  A0*/B0  +  /A0* B0                ;A0 :+: B0
   +  A1*/B1  +  /A1* B1                ;A1 :+: B1
   +  A2*/B2  +  /A2* B2                ;A2 :+: B2
   +  A3*/B3  +  /A3* B3                ;A3 :+: B3
   +  A4*/B4  +  /A4* B4                ;A4 :+: B4
   +  A5*/B5  +  /A5* B5                ;A5 :+: B5
   +  A6*/B6  +  /A6* B6                ;A6 :+: B6
   +  A7*/B7  +  /A7* B7                ;A7 :+: B7


SIMULATION

TRACE_ON A7 A6 A5 A4 A3 A2 A1 A0 NE
         B7 B6 B5 B4 B3 B2 B1 B0

SETF  A7 /A6 /A5 /A4 /A3 /A2 /A1 /A0    ;A7=H, B7=L
     /B7 /B6 /B5 /B4 /B3 /B2 /B1 /B0
SETF /A7 A6                             ;A6=H, B6=L
SETF /A6 A5                             ;A5=H, B5=L
SETF /A5 A4                             ;A4=H, B4=L
SETF /A4 A3                             ;A3=H, B3=L
SETF /A3 A2                             ;A2=H, B2=L
SETF /A2 A1                             ;A1=H, B1=L
SETF /A1 A0                             ;A0=H, B0=L
SETF /A7 /A6 /A5 /A4 /A3 /A2 /A1 /A0    ;A7=L, B7=H
      B7
SETF /B7 B6                             ;A6=L, B6=H
SETF /B6 B5                             ;A5=L, B5=L
SETF /B5 B4                             ;A4=L, B4=H
SETF /B4 B3                             ;A3=L, B3=H
SETF /B3 B2                             ;A2=L, B2=H
SETF /B2 B1                             ;A1=L, B1=H
SETF /B1 B0                             ;A0=L, B0=H
SETF /B0                                ;Test all L's
SETF A7 A6 A5 A4 A3 A2 A1 A0            ;Test all H's
     B7 B6 B5 B4 B3 B2 B1 B0
SETF /A7 A6 /A5 A4 /A3 A2 /A1 A0        ;Test even ones
     /B7 B6 /B5 B4 /B3 B2 /B1 B0
SETF A7 /A6 A5 /A4 A3 /A2 A1 /A0        ;Test odd ones
     B7 /B6 B5 /B4 B3 /B2 B1 /B0

TRACE_OFF
```

Figure 21. Octal Comparator Design Example

Note that the EQL equation, as well as GTR and LES equations, rely upon the XOR function. Often the logic represented by the equations is implemented in two or more devices. There are several PAL devices that provide dedicated XOR capability, and can efficiently implement the logic. These devices, listed in Figure 22, can be used for comparator applications. The PAL32VX10 and the PAL22RX8 offer an architecture with a multiple number of product terms on one side of the XOR gate, which allows a number to be compared against multiple comparands. The design example in Figure 19 shows the XOR implementation of the LES, GTR and EQL functions.

Let us analyze these equations further. The LES and GTR outputs indicate whether one number is greater than or less than another. In fact, these equations can also be judiciously combined to get a comparison of a range of numbers such as A > X > B. Such range comparisons are very useful for address decoder circuits.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16X4	20	8	4	4	8	14
20X10 20X8 20X4	24 24 24	10 10 10	10 8 4	- 2 6	4 4 4	22.2 22.2 22.2
22XP10	24	12	-	10	8	t_{PD} = 20, 30, 40
20XRP10 20XRP8 20XRP6 20XRP4	24 24 24 24	10 10 10 10	10 8 6 4	- 2 4 6	8 8 8 8	30, 22.2, 14 30, 22.2, 14 30, 22.2, 14 30, 22.2, 14
22RX8 32VX10	24 24	14 12	0-8 0-10	8-0 10-0	8 8-16	28.5 25, 22.2

Figure 22. PAL Devices with XOR Capability for Comparators

Range Decoders

Range decoders implemented as address decoders are one of the most commonly used applications of PLDs in digital systems. A good example is the address decoder illustrated earlier on page 2-22, in the PLD Design Methodology section. Range decoders compare a number (address) to a given range of comparands (addresses). One way to arrive at the range decoder Boolean equations is to use the traditional truth table approach. Another way is to use the Boolean equations generated earlier in the comparator section for greater-than and less-than functions. To decode a range of three-bit numbers from B to A, we must compare another number X such that A>X>B. The Boolean equations for the GTR (A>X) and LES (B<X) functions are illustrated below:

```
GTR =  A3 * /X3
     + (A3 :+: /X3) * A2 * /X2
     + (A3 :+: /X3) * (A2 :+: /X2) * A1 * /X1

LES =  X3 * /B3
     + (X3 :+: /B3) * X2 * /B2
     + (X3 :+: /B3) * (X2 :+: /B2) * X1 * /B1
```

Combining these two equations can give us a range signal which will be asserted only when A is greater than X and X is greater than B. The combined Boolean equation follows:

```
RANG = ( A3 * /X3
       + (A3 :+: /X3) * A2 * /X2
       + (A3 :+: /X3) * (A2 :+: /X2) * A1 * /X1 )
       *
        ( X3 * /B3
       + (X3 :+: /B3) * X2 * /B2
       + (X3 :+: /B3) * (X2 :+: /B2) * X1 * /B1 )
```

Using Boolean algebra we get the following equation:

```
RANG =
   (A3 :+: /X3) * (X3 :+: /B3) * (A2 :+: /X2) * A1 * /X1 * X2 * /B2
 + (A3 :+: /X3) * (X3 :+: /B3) * (X2 :+: /B2) * A2 * /X2 * X1 * /B1
 + (A3 :+: /X3) * (A2 :+: /X2) * A1 * /X1 * X3 * /B3
 + (X3 :+: /B3) * (X2 :+: /B2) * A3 * /X3 * X1 * /B1
 + (A3 :+: /X3) * A2 * /X2 * X3 * /B3
 + (X3 :+: /B3) * A3 * /X3 * X2 * /B2
```

The general equation for n-bit comparands can also be obtained by extending these equations.

```
RANG = ( An * /Xn
     + (An :+: /Xn) * An-1 * /Xn-1
     + (An :+: /Xn) * (An-1 :+: /Xn-1) * An-2 * /Xn-2
     + ...
     + ...
     + (An :+: /Xn) * (An-1 :+: /Xn-1) ... (A2 :+: /X2) * A1 * /X1)
   *
      (Xn * /Bn
     + (Xn :+: /Bn) * Xn-1 * /Bn-1
     + (Xn :+: /Bn) * (Xn-1 :+: /Bn-1) * Xn-2 * /Bn-2
     + ...
     + ...
     + ...
     + (Xn :+: /Bn) * (Xn-1 :+: /Bn-1) ... (X2 :+: /B2) * X1 * /B1)
```

The number of product terms required is clearly very large and can easily exceed one hundred for an eight-bit range comparator. Most microprocessors have addresses which exceed 16 bits. In order to fit the design on a PAL device, one commonly used technique is to select the address range defined by A and B such that the range extends from address B+1 to A-1, where A and B+1 are power-of-two numbers. Because the address space is aligned on the power-of-two boundaries, a number of bits of the address comparands will be zero. When implemented in Boolean equations, this substantially reduces the number of product terms required.

The maximum number of product terms required for a three-bit range decoder shown above, with any comparand values, is 28. If the address is chosen from 2 to 3, resulting in A=4 and B=1, then only one product term will be required.

```
RANG = /X3 *  X2
```

Similarly for a range from one to five, B = 1 and A = 6 (a multiple-of-two), and the number of product terms required is only two.

```
RANG = /X3 *  X2
     + X3 * /X2
```

Thus a careful selection of range boundaries allows such logic functions to be implemented easily in PLDs. Such reduction in logic obviously also holds true for discrete implementations. Most address decoders are designed with address ranges with boundaries that are power-of-two numbers, and require few product terms for implementation.

For a power-of-two address range, the comparand A would be a power-of-two; 2, 4 or 8. These are numbers whose least significant bits are all zeros. Similarly, comparand B will be a power-of-two number (minus one); 1, 3, and 7. These are numbers whose least significant bits are all ones. Substituting these in the general equation for range comparators we arrive at:

```
A = 0000100000000000
    n   p          1

B = 0000000000011111
    n          q   1
```

RANG = /Xn * /Xn-1 *...* /Xp *(Xp-1 + Xp-2 + ... +Xq)

This is a general equation for a power-of-two range comparison. In the example on page 2-22, the ROMCS2 signal addresses the range from 100000 to 1FFFFF, in which case B=0FFFFF and A=2FFFFF. Here n=23, p=22 and q=21. The address decode equation for the ROMCS2 signal can be arrived at by substituting:

```
ROMCS2 = /A23 * /A22 * A21
```

This is the same equation as was found from the truth table.

Such designs are very common for address decoder applications. These do not require any XOR gates, and can be implemented in standard combinatorial PLDs with only sum-of-products logic. A list of such PLDs is shown in Figure 20.

Adders/Arithmetic Circuits

Digital systems are designed to carry out a variety of arithmetic instructions on binary numerical data. A good example is the ALU (Arithmetic Logic Unit) used in all digital computers. The basic function of an ALU is that of an adder performing addition on two binary numbers. A binary adder takes two inputs, adds them, and generates the binary sum. A full adder is a one-bit adder with carry-in and carry-out; this is the basic building block of any adding circuit. The truth table of such an adder is shown in Figure 23.

INPUTS			OUTPUTS	
A	B	C_{IN}	Y	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 23. Truth Table for a Full Adder

This truth table is then used to form the Boolean equations in the manner described earlier.

```
Y =  A * /B * /Cin
   + /A * /B *  Cin
   +  A *  B *  Cin
   + /A *  B * /Cin

Cout =  A * Cin
      + A * B
      + B * Cin
```

Bigger binary adders can be made by cascading these full adders. Each carry-out is directed to carry-in for the next stage. Such adders are known as Ripple Adders.

Combinatorial PAL devices are ideal for this purpose, since they also provide internal feedback. Thus one strong consideration in such designs is the internal feedback capability of the device, in addition to other general device selection considerations.

These ripple adders have an advantage in that they can be cascaded to any length possible. But since the carry-out from the least significant bit has to travel all the way to the highest significant bit, this can take a long time, making such large adders inefficient. Adders with built in carry-look-ahead circuitry can save time by simultaneously generating the carry-in signal for all of the bits.

Rewriting the equations for a full adder from above gives:

```
Y0 = A0 :+: B0 :+: Cin
```

where the carry-out signal is:

```
C0 = A0 * B0 + (A0 + B0) * Cin
```

Extending these equations for an n-bit carry-look-ahead adder, we can directly get the following equations:

```
Y0 = A0 :+: B0 :+: Cin

Y1 = A1 :+: B1 :+: C0
```

where

```
C0 = A0 * B0
   +(A0 + B0) * Cin

Y2 = A2 :+: B2 :+: C1
```

where

```
C1 = A1 * B1
   +(A1 + B1)*(A0 * B0)
   +(A1 + B1)*(A0 + B0) * Cin

Y3 = A3 :+: B3 :+: C2
```

where

```
C2 = A2 * B2
   +(A2 + B2)*(A1 * B1)
   +(A2 + B2)*(A1 + B1)*(A0 * B0)
   +(A2 + B2)*(A1 + B1)*(A0 + B0) * Cin
```

In general

```
Yn = An :+: Bn :+: Cn-1

and Cn-1 = An-1 * Bn-1
         +(An-1 + Bn-1)*(An-2* Bn-2)
         + ...
         + ...
         +(An-1 + Bn-1)* ... * (A0 + B0) * Cin
```

and finally the carry-out is:

```
Cn =  An * Bn
   + (An + Bn) * (An-1 * Bn-1)
   + (An + Bn) * (An-1 + Bn-1) * (An-2 * Bn-2)
   + ...
   + ...
   + (An + Bn) * ... * (A0 + B0) * Cin
```

These equations are essentially a combination of the traditional generate and propagate logic for ALU design.

Adder Device Selection Considerations

The number of product terms required for implementing a carry-look-ahead adder is enormous. The carry-out function alone for a four-bit carry-look-ahead adder requires over 36 product terms in the sum-of-products form. For a single-level AND-OR implementation the number of product terms required for the most significant bit Y3 is 28. This will not fit into any of the available standard combinatorial devices. A PLD with this many product terms per output will not only be slow, but will also allow very low silicon utilization.

Special circuitry, known as bit-pair decoding, has been provided on the PAL16X4 to allow implementation of such functions with up to six product terms. Designed primarily for arithmetic operations, the PAL16X4 device has an architecture which implements three different stages of logic. The first is the bit-pair decoding between an input and a feedback. The second is the standard AND-OR array logic. The third is the XOR gate provided on selected outputs.

Although adders are strictly a combinatorial function, the PAL16X4 provides four registered outputs. Not only are these registers used for storing the result, but they are also used temporarily to hold one of the two operands being added. The other operand is provided directly at the inputs. To keep the discussion simple we will not discuss how the first operand is loaded into the registers. This load function can be implemented by using an extra product term on each output.

The logic trick lies in the bit-pair decoding function. All of the bits of the first operand in the registers (A) and the second operand at the inputs (B) are bit-pair decoded. As illustrated in Figure 24, the results of this bit-pair decoding are A + B, A + /B, /A + B, and /A + /B. These outputs are then fed to the AND array as inputs.

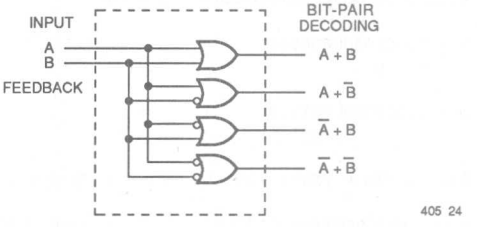
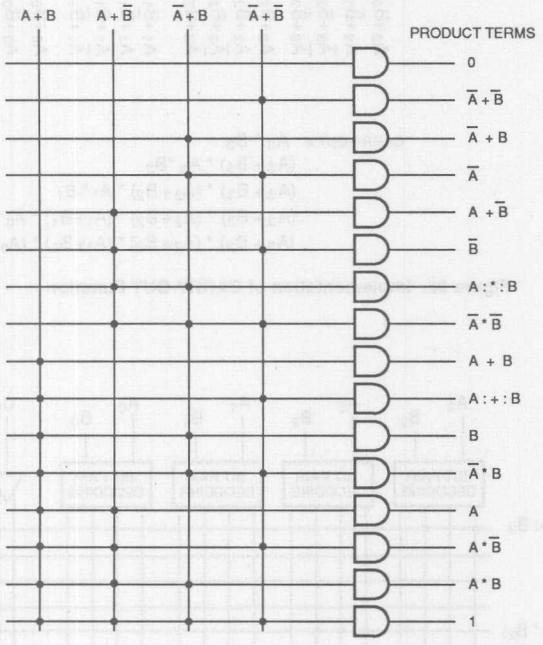


Figure 24. Bit-Pair-Decoding Function

Sixteen AND combinations of these four inputs can then be formed on every product term of the AND-OR array. These are shown in Figure 25, and include the standard true and complements of both the bits as well as XOR, XNOR and various other combinations. This bit pair decoding essentially provides an extra two-level AND-OR logic level before the AND-OR array. The cost as well as extra propagation delay of the extra logic level is minimal, since the array size does not increase.

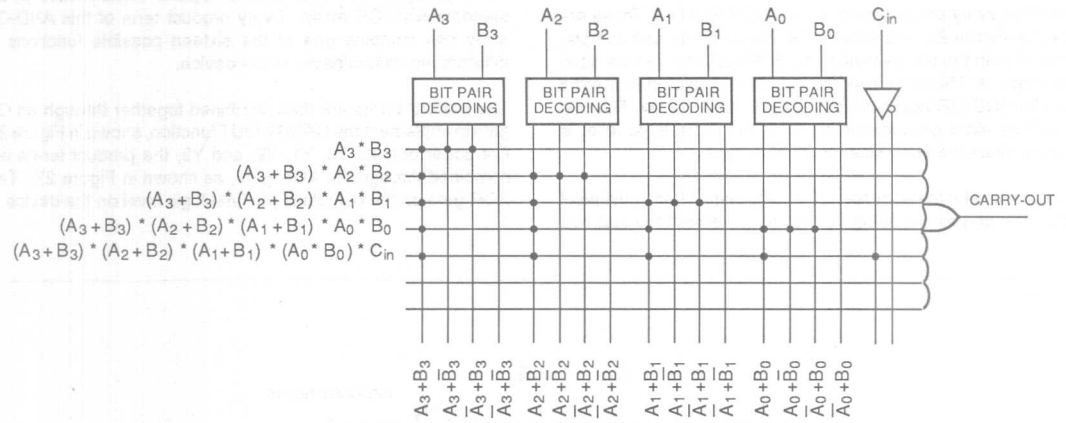
The equations for the adder can obviously benefit from multi-level logic. The bit-pair decoding is used to implement the first two levels of logic. The next level of logic is implemented in the standard AND-OR array. Every product term of this AND-OR array can combine one of the sixteen possible functions of different inputs/feedbacks of the device.

The product terms are then combined together through an OR gate to implement the CARRY-OUT function, shown in Figure 26. For adder outputs Y0, Y1, Y2, and Y3, the product terms are combined through the XOR gate, as shown in Figure 27. This XOR gate on the outputs is the final logic level on the device.



405 25

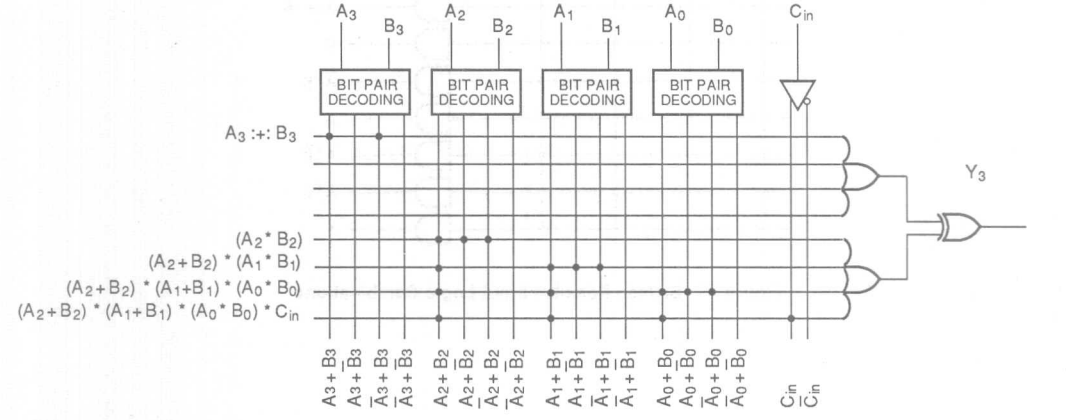
Figure 25. Sixteen Possible Input Logic Combinations



CARRY-OUT = $A_3 * B_3$
$(A_3 + B_3) * A_2 * B_2$
$(A_3 + B_3) * (A_2 + B_2) * A_1 * B_1$
$(A_3 + B_3) * (A_2 + B_2) * (A_1 + B_1) * A_0 * B_0$
$(A_3 + B_3) * (A_2 + B_2) * (A_1 + B_1) * (A_0 * B_0) * C_{in}$

405 26

Figure 26. Implementation of CARRY-OUT Function



$Y_3 = A_3 :+: B_3$

$:+: ((A_2 * B_2)$
$+(A_2 + B_2) * (A_1 * B_1)$
$+(A_2 + B_2) * (A_1 + B_1) * (A_0 * B_0)$
$+(A_2 + B_2) * (A_1 + B_1) * (A_0 * B_0) * C_{in})$

405 27

Figure 27. Relationship Between Adder Boolean Equation and PAL16X4 Device Logic

Thus the most significant bit of the output can be implemented using only five product terms; the carry-out function can be implemented using only six product terms. All of this is possible because the PAL16X4 offers a modified programmable array with hardware dedicated to bit-pair decoding at the inputs and an XOR gate at the outputs.

Figure 28 illustrates an example of one such adder, which also includes the logic required for loading the registers with one of the operands.

```
PAL16X4                                    PAL DESIGN SPECIFICATION
PNEW                                         J. ENGINEER 9/15/87
ADDER
MMI SUNNYVALE, CALIFORNIA

CLK LOAD ADD B0 B1 B2 B3  NC NC GND
CIN NC   NC  A3 A2 A1 A0  G  P  VCC


;DESCRIPTION

;THE DEVICE FUNCTIONS AS AN ADDER. IT FIRST STORES ONE OPERAND IN THE FOUR-
;BIT OUTPUT REGISTER UNDER THE CONTROL OF LOAD SIGNAL. USING FEEDBACK AND A
;SECOND OPERAND AT THE INPUTS, IT ADDS THE TWO UNDER CONTROL OF SIGNAL ADD.
;THE RESULTS ARE STORED BACK IN THE OUTPUT REGISTERS. THE DEVICE ALSO
;GENERATES THE PROPAGATE AND GENERATE SIGNALS WHICH ALLOW A NUMBER OF
;SIMILAR DEVICES TO BE CASCADED.


IF(VCC)   P = (A3 + B3)       ; PROPAGATE SIGNAL
            * (A2 + B2)
            * (A1 + B1)
            * (A0 + B0)

IF(VCC)  G  = (A3*B3)         ; GENERATE SIGNAL
            + (A3 + B3) * (A2*B2)
            + (A3 + B3) * (A2 + B2) * (A1*B1)
            + (A3 + B3) * (A2 + B2) * (A1 + B1) * (A0*B0)

CARRY0.EQU  ADD * CIN

CARRY1.EQU  ADD * (A0*B0)
          + ADD * (A0 + B0) * CIN

CARRY2.EQU  ADD * (A1*B1)
          + ADD * (A1 + B1) * (A0*B0)
          + ADD * (A1 + B1) * (A0 + B0) * CIN

CARRY3.EQU  ADD * (A2*B2)
          + ADD * (A2 + B2) * (A1*B1)
          + ADD * (A2 + B2) * (A1 + B1) * (A0*B0)
          + ADD * (A2 + B2) * (A1 + B1) * (A0 + B0) * CIN


       /A3 := (/A3)*/LOAD * /ADD     ;HOLD  REG A3
            + (/B3)* LOAD            ;LOAD  REG B3
            + (A3:*:B3)*AD           ;ADD A3 AND B3
           :+: CARRY3

       /A2 := (/A2)*/LOAD * /ADD     ;HOLD  REG A2
            + (/B2)* LOAD            ;LOAD  REG B2
            + (A2:*:B2)*ADD          ;ADD A2 AND B2
           :+: CARRY2

       /A1 := (/A1)*/LOAD * /ADD     ;HOLD  REG A1
            + (/B1)* LOAD            ;LOAD  REG B1
            + (A1:*:B1)*ADD          ;ADD A1 AND B1
           :+: CARRY1

       /A0 := (/A0)*/LOAD * /ADD     ;HOLD  REG A0
            + (/B0)* LOAD            ;LOAD  REG B0
            + (A0:*:B0)*ADD          ;ADD A0 AND B0
           :+: CARRY0

FUNCTION TABLE

; NOT INCLUDED HERE
```

Figure 28. A Four-Bit Adder Implemented in a PAL16X4

Latches

PAL devices are often used to implement latches. One of the most common uses for latches is as temporary storage for data or addresses. PLD-based latches are often used in address decoders to assert the decoded signal for long durations. These latches are also very useful for asynchronous digital designs, and are used often for control and arbitration functions.

A latch is essentially a simple combinatorial circuit in which the output is a function of inputs and feedback. The most commonly used latch is the D-type latch. When the control signal latch-enable (LEN) is HIGH, the latch is in the "transparent mode" and the input signal /D is available at the outputs. When the LEN signal is LOW, the input data is latched on the outputs and is retained until LEN goes back HIGH. In a typical address decoder, the input will be a combination of various address signals, decoded as explained earlier for range comparators. The latching signal in most microprocessors is called AS (address strobe) or ALE (address latch enable).

The truth table for a latch can be derived directly from this functional description, and is shown in Figure 29.

INPUTS		OUTPUTS
/D	LEN	/Q
0	1	0
1	1	1
X	0	/Q (previous)

Figure 29. Truth Table of a Simple Latch

The Boolean equations for this latch can be directly derived from the truth table:

```
/Q =  /D * LEN
    + /Q * /LEN
```

The logic implementation for this latch is shown in Figure 30.

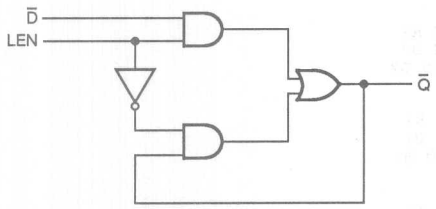


Figure 30. A Transparent Latch

Hazards

Even when a combinatorial circuit has been designed correctly, it may still have erroneous outputs due to "hazards". Hazards exist because physical circuits do not behave ideally. Combinatorial complementary output functions based on the same inputs are prime candidates for such hazards. As the input changes, the two outputs will not respond simultaneously and change state at the same instant. Although this will not change the steady-state output of the circuit, it may cause a spurious pulse or a "glitch". Such hazards are even more dangerous in latches, where the glitch can cause incorrect data to be latched.

There are two types of hazards, static and dynamic. Static hazards occur when the steady-state output of combinatorial logic is not supposed to change due to an input transition, but a momentary change does occur. Such a glitch can be further classified as a static 1 or a static 0 hazard as shown in Figure 31.

Figure 31. Static Hazards

Dynamic hazards involve situations where the the steady-state output is supposed to change due to an input transition. The hazard occurs when the transient output changes several times before settling. Figure 32 shows dynamic hazards.

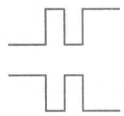


Figure 32. Dynamic Hazards

A Karnaugh map is a very good way of detecting hazard conditions. When trying to detect a static 0 or static 1 hazard, only the mapping of the zeros and the ones, respectively, are required. For example, the latch equations in Figure 30 can be mapped to a Karnaugh map shown in Figure 33. The relationship between the Karnaugh mapping and the Boolean equation product terms is also illustrated.

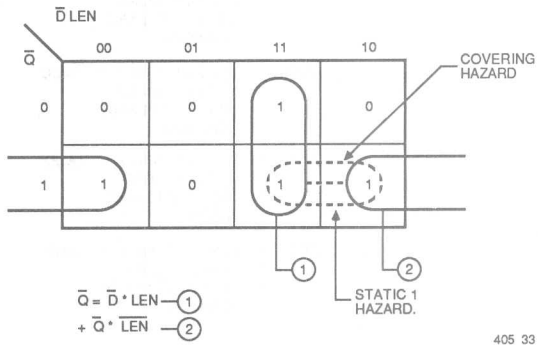


Figure 33. Karnaugh Map for Transparent Latch Design

The possibility of a hazard exists when the signal LEN changes. Initially, when $\overline{D}$ and LEN are HIGH, the output $\overline{Q}$ is also HIGH. When LEN switches to LOW, it is possible for the output to go LOW momentarily. This is because when LEN goes LOW the first product term is disabled, and to maintain the output HIGH the second product term should be enabled exactly at the same instant. Due to the uneven gate delays or routing conditions on the board, these two events will not take place simultaneously. This is a static 1 hazard. It can also be identified directly in the Karnaugh map by the two adjacent but disjoint sets of ones, grouped together to form a product term each.

The hazard conditions can be easily avoided in the PLDs by providing an extra cover product term. This product term is shown with a dotted line in the Karnaugh map. This third product term will keep the output asserted during the transition of the LEN signal, when the control changes from the first product term to the second. The modified Boolean equation is shown above.

```
/Q = /D * LEN
   + /Q * /LEN
   + /D * /Q   (Cover product term)
```

Devices on which latches are implemented need to provide output feedback. All devices with I/O pins provide this necessary feedback. The only other consideration for selecting a device would be the provision of sufficient number of product terms for addressing the needs of glitch-free and testable design.

A complete latch design implemented in a PAL20L10 is shown in Figure 34. Although the design is now glitch-free, the cover product term introduces further complications in the testability of this circuit. This design does not have the additional circuitry to address the testability requirements. Details on designing for testability are discussed on page 3-108.

2

```
TITLE      CLEAN OCTAL LATCH
PATTERN    P7096
REVISION   A
AUTHOR     VINCENT COLI
COMPANY    MONOLITHIC MEMORIES, INC.
DATE       03/10/83

CHIP OCT_LAT PAL20L10

 G   NC D0 D1 D2 D3 D4 D5 D6 D7 NC GND
/OC  NC Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0 NC VCC

;DESCRIPTION

;THIS PAL16L8 IMPLEMENTS AN 8-BIT LATCH FUNCTION WITH THREE-STATE OUTPUTS.
;THE LATCH PASSES EIGHT BITS OF DATA (D7-D0) TO THE EIGHT OUTPUTS (Q7-Q0)
;WHEN LATCH ENABLE IS TRUE (G=HIGH).  THE DATA IS LATCHED WHEN LATCH ENABLE
;IS FALSE (G=LOW).  THE OUTPUTS WILL BE DISABLED (HI-Z) WHEN OUTPUT ENABLE
;IS TRUE (/OC=TRUE) REGARDLESS OF ANY OTHER INPUTS.


;/OC   G   D7-D0   Q7-Q0     COMMENTS
;-------------------------------------
; H    X     X       Z       HI-Z
; L    L     X       Q       LATCH OUTPUT
; L    H     D       D       LOAD LATCH
;-------------------------------------


;THIS DESIGN SHOWS HOW TO IMPLEMENT A "CLEAN" LATCH SINCE THERE ARE NO
;OUTPUT GLITCHES AS THE DEVICE CHANGES STATE.
```

Figure 34. Clean Octal Latch Design

```
EQUATIONS

        /Q0 =   G*/D0       ;LOAD LATCH (Q0)
            +  /G*/Q0       ;LATCH OUTPUT
            + /D0*/Q0       ;COVER ALWAYS HIGH HAZARD

        Q0.TRST = OC        ;OUTPUT ENABLE

        /Q1 =   G*/D1       ;LOAD LATCH (Q1)
            +  /G*/Q1       ;LATCH OUTPUT
            + /D1*/Q1       ;COVER ALWAYS HIGH HAZARD

        Q1.TRST = OC        ;OUTPUT ENABLE

        /Q2 =   G*/D2       ;LOAD LATCH (Q2)
            +  /G*/Q2       ;LATCH OUTPUT
            + /D2*/Q2       ;COVER ALWAYS HIGH HAZARD

        Q2.TRST = OC        ;OUTPUT ENABLE

        /Q3 =   G*/D3       ;LOAD LATCH (Q3)
            +  /G*/Q3       ;LATCH OUTPUT
            + /D3*/Q3       ;COVER ALWAYS HIGH HAZARD

        Q3.TRST = OC        ;OUTPUT ENABLE

        /Q4 =   G*/D4       ;LOAD LATCH (Q4)
            +  /G*/Q4       ;LATCH OUTPUT
            + /D4*/Q4       ;COVER ALWAYS HIGH HAZARD

        Q4.TRST = OC        ;OUTPUT ENABLE

        /Q5 =   G*/D5       ;LOAD LATCH (Q5)
            +  /G*/Q5       ;LATCH OUTPUT
            + /D5*/Q5       ;COVER ALWAYS HIGH HAZARD

        Q5.TRST = OC        ;OUTPUT ENABLE

        /Q6 =   G*/D6       ;LOAD LATCH (Q6)
            +  /G*/Q6       ;LATCH OUTPUT
            + /D6*/Q6       ;COVER ALWAYS HIGH HAZARD

        Q6.TRST = OC        ;OUTPUT ENABLE

        /Q7 =   G*/D7       ;LOAD LATCH (Q7)
            +  /G*/Q7       ;LATCH OUTPUT
            + /D7*/Q7       ;COVER ALWAYS HIGH HAZARD

        Q7.TRST = OC        ;OUTPUT ENABLE


; SIMULATION NOT INCLUDED HERE
```

Figure 34. Clean Octal Latch Design (Cont'd.)

Registered Logic Design

In the previous section we discussed combinatorial designs, circuits whose outputs are totally independent of any system clock. In this section we will discuss sequential circuits, where outputs store their previous values until a new clock is applied. The storage elements which retain the previous output values are called flip-flops. A bank of these flip-flops forms a register, although individual flip-flops are often called registers. Monolithic Memories and Advanced Micro Devices offer a number of PLDs which provide registered outputs. The table in Figure 1 lists all of the PLDs which provide registered outputs.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16R8	20	8	8	-	8	55.5, 37, 25, 16
16R6	20	8	6	2	8	55.5, 37, 25, 16
16R4	20	8	4	4	8	55.5, 37, 25, 16
16RP8	20	8	8	-	8	22.2
16RP6	20	8	6	2	8	22.2
16RP4	20	8	4	4	8	22.2
16RA8	20	8	0-8	8-0	4	20
16X4	20	8	4	4	8	14
23S8	20	9	4	4	8-12	33.3, 28.5
20R8	24	12	8	-	8	37, 25, 16
20R6	24	12	6	2	8	37, 25, 16
20R4	24	12	4	4	8	37, 25, 16
20X10	24	10	10	-	4	22.2
20X8	24	10	8	2	4	22.2
20X4	24	10	4	6	4	22.2
20RP10	24	10	10	-	8	37, 25
20RP8	24	10	8	2	8	37, 25
20RP6	24	10	6	4	8	37, 25
20RP4	24	10	4	6	8	37, 25
20XRP10	24	10	10	-	8	30, 22.2, 14
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
20RS10	24	10	10	-	8-16	19.2
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
32R16	40	16	16	-	8-16	16
PLS167	24	14	6	-	Total 48	33
PLS168	24	12	8	-	Total 48	33
PLS105	28	16	8	-	Total 48	37

Figure 1. Registered PLDs

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
CMOS						
C16R8	20	8	8	-	8	28.5
C16R6	20	8	6	2	8	28.5
C16R4	20	8	4	4	8	28.5
C20R8	24	12	8	-	8	20, 15.3
C20R6	24	12	6	2	8	20, 15.3
C20R4	24	12	4	4	8	20, 15.3
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
10H20EV/EG8	24	12	0-8	8-0	8-12	125
10020EV/EG8	24	12	0-8	8-0	8-12	125

Figure 1. Registered PLDs (Cont'd.)

Before we discuss purely registered designs, let us take a look at designs which combine both registered and combinatorial portions. Registered and combinatorial outputs are often mixed on a single device. There can be two distinct designs, one registered and one combinatorial (often glue logic) combined on a single device for higher integration. There may also be a design requirement where registered outputs need to be decoded using combinatorial logic.

There are a number of devices which provide both registered and combinatorial outputs. Some devices provide programmable register bypass, which allows outputs to be programmed as registered or combinatorial. Figure 2 shows a list of PLDs with both registered and combinatorial outputs. Figure 3 shows a list of PLDs with programmable register bypass.

In most design software packages the output registers are signified by the ":=" assignment symbol, as opposed to the "=" sign for a combinatorial output. This helps easily identify registers in each equation. In devices which provide outputs configurable as either registered or combinatorial, this sign is also used by the software to configure the outputs.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16R6	20	8	6	2	8	55.5, 37, 25, 16
16R4	20	8	4	4	8	55.5, 37, 25, 16
16RP6	20	8	6	2	8	22.2
16RP4	20	8	4	4	8	22.2
16RA8	20	8	0-8	8-0	4	20
16X4	20	8	4	4	8	14
23S8	20	9	4	4	8-12	33.3, 28.5
20R6	24	12	6	2	8	37, 25, 16
20R4	24	12	4	4	8	37, 25, 16
20X8	24	10	8	2	4	22.2
20X4	24	10	4	6	4	22.2
20RP8	24	10	8	2	8	37, 25
20RP6	24	10	6	4	8	37, 25
20RP4	24	10	4	6	8	37, 25
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
32R16	40	16	16	-	8-16	16
CMOS						
C16R6	20	8	6	2	8	28.5
C16R4	20	8	4	4	8	28.5
C20R6	24	12	6	2	8	20, 15.3
C20R4	24	12	4	4	8	20, 15.3
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
Latched						
10H20G8	24	12	0-8	8-0	4-8	N.A. (latch)
10H/020EG8	24	12	0-8	8-0	8-12	N.A. (latch)
10H/020EV8	24	12	0-8	8-0	8-12	125

Figure 2. PLDs with Both Registered and Combinatorial Outputs

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16RA8	20	8	0-8	8-0	4	20
23S8	20	9	4	4	8-12	33.3, 28.5
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
32R16	40	16	16	-	8-16	16
CMOS						
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
Latched						
10H20G8	24	12	0-8	8-0	4-8	N.A. (latch)
10H/020EG8	24	12	0-8	8-0	8-12	N.A. (latch)
10H/020EV8	24	12	0-8	8-0	8-12	125

Figure 3. PLDs with Programmable Register Bypass

General Device Selection Considerations

The same set of general device selection considerations discussed in the PLD design methodology section (page 2-21) apply to registered designs. The list of items which must be considered is repeated in Figure 4 for convenience. A device can be conveniently selected from the table of registered PLDs (Figure 1) based upon the specific input and output requirements.

1. Number of input pins
2. Number of output pins
3. Number of I/O pins
4. Device speed
5. Device power requirements
6. Number of registers
7. Number of product terms
8. Output polarity control

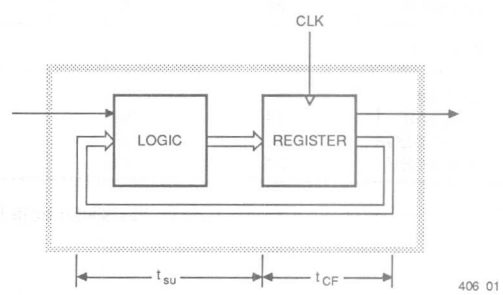
Figure 4. General Device Selection Considerations

Maximum Frequency

For registered designs, speed is a parameter which needs careful consideration. Most combinatorial designs use the propagation delay (t_{PD}) for ensuring that enough time is allowed for the data from the inputs to appear at the outputs. In registered designs the effects of the clock must be taken into account. This is reflected in the maximum frequency (f_{MAX}) parameter. The flexibility inherent in PLD design provides a choice of configurations from which different f_{MAX} parameters can be calculated.

In the first type of design, the PLD is used for a stand-alone registered design. In order to decide the next logic level of the registers, the present logic level needs to be available at the inputs of the registers before they are clocked (Figure 5). Under these conditions the clock period is limited by the internal delay from the flip-flop outputs through the internal feedback and logic to the flip-flops inputs ($t_{su}+t_{CF}$). For this configuration:

$f_{MAX}=1/(t_{su}+t_{CF})$: *designated the internal* f_{MAX}.



f_{MAX} INTERNAL: $1/(t_{su}+t_{CF})$

Figure 5. Internal f_{MAX}

The second type of system configuration is when a number of logic devices with registers, including PLDs, are clocked with a common clock. This is probably the most prevalent configuration. In this case the registered outputs are sent off-chip back to the device inputs or to the inputs of a second device. The slowest path required to set up an input (Figure 6) is the sum of the clock-to-output delay and the input setup time for the external signals ($t_{su} + t_{CLK}$). The same speed grade devices are typically clocked together, which means that the t_{su} and t_{CLK} of all the devices will usually be the same.

Thus for this configuration:

$$f_{MAX} = 1/(t_{su} + t_{CLK})$$

This f_{MAX} is the maximum frequency when external feedback is used or when several equivalent-speed devices are used. This f_{MAX} is designated the *external* f_{MAX}. The tables for registered devices show this f_{MAX} parameter.

The third type of design is a simple data path application. In this case, input data is presented to the flip-flop and clocked; no feedback is employed (Figure 7). In this case, the period is limited by the sum of data setup time and data hold time ($t_{su}+t_h$). However, the minimum clock period ($t_{WH} + t_{WL}$) is usually a stricter limit. Thus the third f_{MAX}, designated "f_{MAX}, *no feedback*" will be the lesser of:

$1/(t_{su} + t_h)$ $\quad$ $1/(t_{WH} + t_{WL})$

Flip-flop Types

There are four basic types of flip-flops; S-R, J-K, T and the popular D-type. Details on the various types of flip-flops are provided in the logic reference (page 6-8). In this section we will assume that the reader is familiar with these flip-flops.

Almost all registered PLDs provide the basic D-type flip-flops. D-type flip-flops are the simplest to design with and will be used throughout this section. Some PLDs provide the capability of configuring output registers as either D, T, J-K or S-R. Configurable flip-flops in some cases can reduce the number of product terms required for certain designs. The effect of the configurable flip-flops will be discussed wherever relevant.

Synchronous vs Asynchronous

Registered designs can be easily classified into two categories; synchronous and asynchronous. In synchronous designs the clock inputs of all the registers are tied together to a common clock. This is the most commonly used type of registered design. With asynchronous designs, the flip-flops' clock inputs may not be tied together, and the clocks may be gated or even driven by other flip-flops. We will first discuss synchronous registered designs and then asynchronous registered designs.

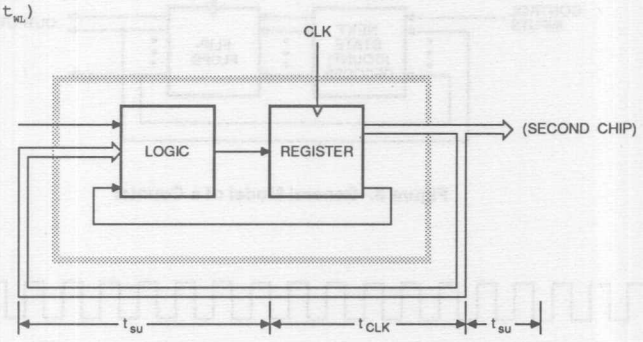


Figure 6. External f_{MAX}

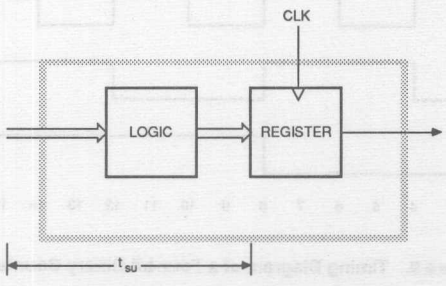


Figure 7. f_{MAX} with No Feedback

Synchronous Registered Designs

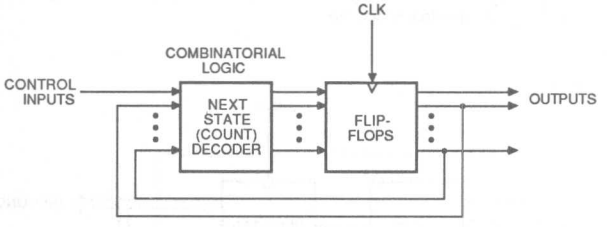
Synchronous registered designs are used for two major functions: data handling and control. Registered synchronous designs for data handling include counters and shift registers. There are various types of counters, among them binary counters, modulo counters, Johnson counters, and Gray-code counters. These counters are differentiated by the sequence of values through which the counter travels. A binary counter is the simplest form of a counter, and is used most often for data functions. Any system requiring a regular count uses a binary counter. Modulo, Gray-code and Johnson counters are also used for control.

All counters are actually subsets of a larger class of digital designs called state machines. State machines are discussed in detail in the next chapter of this handbook.

Counters

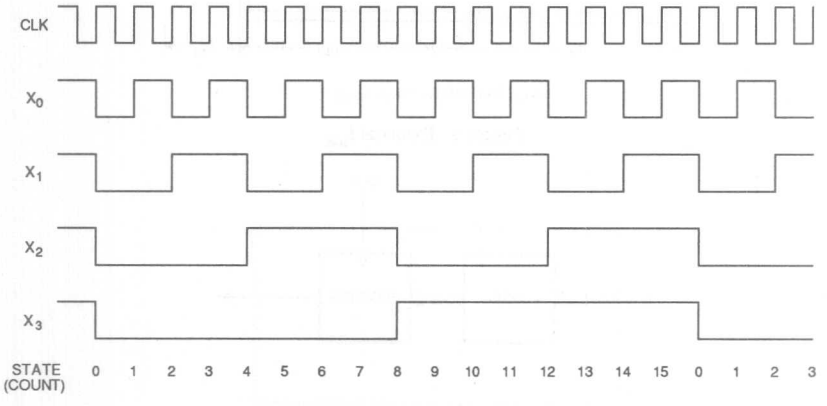
Counters are the most commonly used sequential circuits. A set of registers which cycles through a predetermined, unvarying sequence is called a counter. A general model of a synchronous counter is illustrated in Figure 8. This shows a common clock to all the flip-flops, whose outputs are fed back to a combinatorial logic array called the next-state (count) decoder. The next count is generated by this logic based upon the present count and control inputs. Most PLDs use the standard sum-of-products form of array for this logic.

The relationship between a four-bit counter and its signal timing diagram is illustrated in Figure 9. The counters can also be represented by state diagrams (Figure 10). The state diagrams are bubble-and-arrow diagrams, with each bubble representing a count value and each arrow a transition from one count to the next. More detail on state diagrams is given in the next chapter on state machine design. For counters, the state diagrams are a convenient representation tool and will be used in the discussion when necessary.



406 04

Figure 8. General Model of a Counter



406 05

Figure 9. Timing Diagram of a Four-bit Binary Counter

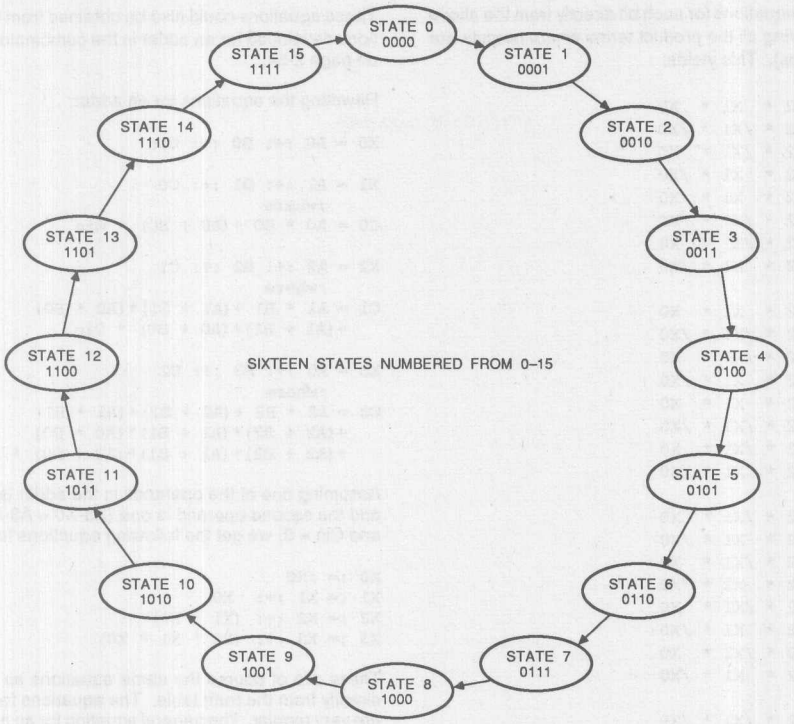


Figure 10. State Diagram of a Four-bit Binary Counter

Binary Counters

Let us examine a four-bit binary counter. The truth table (also called the transition table) for such a counter is given in Figure 11. The table lists the next state values of all the output registers based upon their present values.

PRESENT STATE				NEXT STATE			
X3	X2	X1	X0	X3	X2	X1	X0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

Figure 11. The Truth Table for a Four-bit Binary Counter

We derive Boolean equations for each bit directly from the above truth table by collecting all the product terms where outputs are asserted HIGH (ones). This yields:

```
X3 :=   /X3 *  X2 *  X1 *  X0
      +  X3 * /X2 * /X1 * /X0
      +  X3 * /X2 * /X1 *  X0
      +  X3 * /X2 *  X1 * /X0
      +  X3 * /X2 *  X1 *  X0
      +  X3 *  X2 * /X1 * /X0
      +  X3 *  X2 * /X1 *  X0
      +  X3 *  X2 *  X1 * /X0

X2 :=   /X3 * /X2 *  X1 *  X0
      + /X3 *  X2 * /X1 * /X0
      + /X3 *  X2 * /X1 *  X0
      + /X3 *  X2 *  X1 * /X0
      +  X3 * /X2 *  X1 *  X0
      +  X3 *  X2 * /X1 * /X0
      +  X3 *  X2 * /X1 *  X0
      +  X3 *  X2 *  X1 * /X0

X1 :=   /X3 * /X2 * /X1 *  X0
      + /X3 * /X2 *  X1 * /X0
      + /X3 *  X2 * /X1 *  X0
      + /X3 *  X2 *  X1 * /X0
      +  X3 * /X2 * /X1 *  X0
      +  X3 * /X2 *  X1 * /X0
      +  X3 *  X2 * /X1 *  X0
      +  X3 *  X2 *  X1 * /X0

X0 :=   /X3 * /X2 * /X1 * /X0
      + /X3 * /X2 *  X1 * /X0
      + /X3 *  X2 * /X1 * /X0
      + /X3 *  X2 *  X1 * /X0
      +  X3 * /X2 * /X1 * /X0
      +  X3 * /X2 *  X1 * /X0
      +  X3 *  X2 * /X1 * /X0
      +  X3 *  X2 *  X1 * /X0
```

These Boolean equations are for devices with active-HIGH outputs. These equations can be inverted for devices with active-LOW outputs. The Boolean equations for active-LOW devices can also be directly derived from the truth table by collecting all the product terms where the active-LOW outputs (zeros) are asserted.

Manipulating the equations with Boolean algebra we obtain the Boolean logic equations:

```
X0 := /X0
X1 := X1 :+: X0
X2 := X2 :+: (X1 * X0)
X3 := X3 :+: (X2 * X1 * X0)
```

Similarly for active-LOW output devices
(since /(A :+: B) = /A :+: B):

```
/X0 := X0
/X1 := /X1 :+: X0
/X2 := /X2 :+: (X1 * X0)
/X3 := /X3 :+: (X2 * X1 * X0)
```

These equations could also be obtained from the Boolean equations developed for an adder in the combinatorial design section on page 2-54.

Rewriting the equations for an adder:

```
X0 = A0 :+: B0 :+: Cin

X1 = A1 :+: B1 :+: C0
     ;where
C0 = A0 * B0 +(A0 + B0) * Cin

X2 = A2 :+: B2 :+: C1
     ;where
C1 = A1 * B1 +(A1 + B1)*(A0 * B0)
   +(A1 + B1)*(A0 + B0) * Cin

X3 = A3 :+: B3 :+: C2
     ;where
C2 = A2 * B2 +(A2 + B2)*(A1 * B1)
   +(A2 + B2)*(A1 + B1)*(A0 * B0)
   +(A2 + B2)*(A1 + B1)*(A0 + B0) * Cin
```

Assuming one of the operands in the adder is the number itself and the second operand is one (X3-X0 = A3-A0, B3-B0 = 0001 and Cin = 0) we get the following equations for a counter:

```
X0 := /X0
X1 := X1 :+:  X0
X2 := X2 :+: (X1 * X0)
X3 := X3 :+: (X2 * X1 * X0)
```

These are of course the same equations as the ones derived directly from the truth table. The equations for a binary counter are very regular. The general equation for an n-bit binary counter can be directly expressed:

Xn := Xn :+: (Xn–1 * Xn–2 ... X0)

For devices with active-LOW outputs the general Boolean equations can be derived by inverting both sides of the equation:

/Xn := /Xn :+: (Xn–1 * Xn–2 ... X0)

These equations represent a binary UP counter. Counting backwards for a DOWN counter, the Boolean equations can be similarly generated, either from the truth table or from the adder Boolean equations. The general equation for a DOWN counter is:

Xn := Xn :+: (/Xn–1 * /Xn–2 ... /X0)

This equation is for active-HIGH outputs. For active-LOW output devices the Boolean equation for a DOWN counter is:

/Xn := /Xn :+: (/Xn–1 * /Xn–2 ... /X0)

A four-bit binary counter which combines both UP and DOWN capability is illustrated in Figure 12. This design is implemented in a registered PLD and requires at least six product terms for the most significant output bit. The design logic has been implemented in sum-of-products form. It uses a control input (UP) which selects between the "count up" and "count down" product terms, allowing the counter to count up or down.

Further control functions can be added to these counter equations directly either at the truth-table stage or in the equations. For example, a load data function is required in most counters. This allows registers to be loaded with a count under the control of another input signal (LOAD). When the LOAD signal is HIGH the counter is loaded with the input data, and when the LOAD signal is LOW the counting is resumed.

```
Title     4Bit_Counter
Pattern   4cnt.pds
Revision  B
Author    Mehrnaz Hada, Bill Hollon, Ali Sebt
Company   Monolithic Memories Inc. Santa Clara, CA
Date      1/14/85

CHIP 4BitCoun PAL16RP4

CLK UP AI BI CI DI CLR LOAD NC GND
/OC NC NC D C B A NC NC VCC

;The 4-bit counter counts up or down and has the clear and
;load capability. The clear operation overrides count and
;load. The counter counts up when CLR=low, LOAD=low, and
;UP=high. It counts down whenever CLR=low, LOAD=low, and
;UP=low

EQUATIONS

A   := /A*/B*/C*/D*/UP*/LOAD*/CLR      ;When CLR=1, A=0.
     + /A* B* C* D* UP*/LOAD*/CLR      ;Else it will count
     +  A* B*      /UP*/LOAD*/CLR      ;UP or DOWN.
     +  A*/B*    D*    /LOAD*/CLR
     +  A*   /C*    UP*/LOAD*/CLR
     +  A*    C*/D*    /LOAD*/CLR
     +                  LOAD*/CLR* AI  ;New value is loaded
                                       ;when LOAD=1, CLR=0.

B   :=    /B*/C*/D*/UP*/LOAD*/CLR      ;When CLR=1, B=0.
     +    /B* C* D* UP*/LOAD*/CLR      ;Else it will count.
     +     B* C*/D*    /LOAD*/CLR
     +     B*/C*    UP*/LOAD*/CLR
     +     B*    D*/UP*/LOAD*/CLR
     +                  LOAD*/CLR* BI  ;New value is loaded
                                       ;when LOAD=1, CLR=0.

C   :=       /C*/D*/UP*/LOAD*/CLR      ;When CLR=1, C=0.
     +       /C* D* UP*/LOAD*/CLR      ;Else it will count.
     +        C*/D* UP*/LOAD*/CLR
     +        C* D*/UP*/LOAD*/CLR
     +                  LOAD*/CLR* CI  ;New value is loaded
                                       ;when LOAD=1, CLR=0.

D   :=          /D*    /LOAD*/CLR     ;Count
     +                  LOAD*/CLR* DI  ;New value is loaded
                                       ;when LOAD=1, CLR=0.
```

Figure 12. Design File for a Four-bit Counter

```
SIMULATION

TRACE_ON  AI BI CI DI LOAD CLR UP A B C D

SETF LOAD /CLR AI BI CI DI OC          ;Load all registers
CLOCKF CLK                             ;to HIGH and count up

SETF CLR                               ;Clear all registers
CLOCKF CLK

SETF /CLR UP /LOAD                     ;Start Counting up

 FOR I:= 1 TO 16 DO                    ;Count up 16 clock
  BEGIN                                ;cycles
    CLOCKF CLK
  END

SETF LOAD /CLR /UP AI BI CI DI         ;Load all registers
CLOCKF                                 ;to HIGH and count
SETF /LOAD                             ;down
  FOR I:= 1 TO 16 DO                   ;Count down 16 clock
   BEGIN                               ;cycles
     CLOCKF CLK
   END

SETF LOAD CLR AI /BI CI /DI            ;Test setting LOAD
CLOCKF CLK                             ;and CLR on at the
                                       ;same time.
SETF /OC

TRACE_OFF
```

Figure 12. Design File for a Four-bit Counter (Cont'd.)

Binary Counter Device Selection Considerations

One major device selection consideration is the logic requirement. The binary counter Boolean equations make use of exclusive-OR functions in the output. In most of the registered PLDs shown in Figure 1, the XOR functions are implemented in their sum-of-products logic form. This usually requires a large number of product terms. We have seen a design example for a four-bit UP/DOWN binary counter which requires up to six product terms. Most standard PAL devices provide eight product terms per output. However, for larger counters, a large number of product terms is required. Figure 13 shows a list of PLDs which provide more than eight product terms per output.

Some PLDs provide a dedicated XOR gate on the outputs (Figure 14). This allows an AND-OR-XOR implementation of the Boolean logic, and consequently requires fewer product terms. A nine-bit binary counter design is illustrated in Figure 15. This design has been implemented in the PAL20X10, which provides an XOR gate. This design cannot be implemented in standard PAL devices.

Cascading Binary Counters

Situations are occasionally encountered in digital system designs where very long counters are required. Few PLDs provide greater than ten outputs to implement such counters. Binary counters can be easily cascaded into two or more devices to construct such large counters. The design of long counters is very simple. These are designed as simple binary counters with a count enable control. The less significant counters generate an extra output signal at the penultimate count. These signals are ANDed together to form the count enable signal for the higher-order counter. For a down counter the reverse scheme is implemented.

Cascading counters is a lot simpler than cascading adders because the carry-look-ahead circuitry is not required. The only thing to remember is that the more significant counter toggles only when the penultimate count of all of the less significant counters is reached.

Notice that the nine-bit counter has a built-in mechanism for cascading. This count enable output signal is represented by CO and is easily implemented in the PLD. This is just another example of how such modifications are easy to handle in a PLD design, unlike standard SSI/MSI designs.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
23S8	20	9	4	4	8-12	33.3, 28.5
20RS10	24	10	10	-	8-16	19.2
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
PLS167	24	14	6	-	Total 48	33
PLS168	24	12	8	-	Total 48	33
PLS105	28	16	8	-	Total 48	37
CMOS						
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
10H20EV/EG8	24	12	0-8	8-0	8-12	125
10020EV/EG8	24	12	0-8	8-0	8-12	125

Figure 13. Registered PLDs with More Than Eight Product Terms Per Output

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16X4	20	8	4	4	8	14
20X10	24	10	10	-	4	22.2
20X8	24	10	8	2	4	22.2
20X4	24	10	4	6	4	22.2
20XRP10	24	10	10	-	8	30, 22.2, 14
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
22RX8	24	14	0-8	8-0	8	28.5
32VX10	24	12	0-10	10-0	8-16	25, 22.2

Figure 14. Registered PLDs for Binary Counter Design with XOR Gate Output

```
Title     9BitCounter
Pattern   9BitCnt.pds
Revision  A
Author    Mehrnaz Hada
Company   Monolithic Memories Inc., Santa Clara, CA
Date      1/28/85


;The 9-bit synchronous counter has parallel load, increment,
;and hold capabilities. The carry out pin (/CO) shows how to
;implement a carry out using a register by anticipating one
;count before the terminal count if counting and the terminal
;count if loading.

;Operations Table

;      /OC    CLK    /LD    D8-D0    Q8-Q0      Operation
;      ------------------------------------------------
;       H      X      X       X        Z        HI-Z
;       L      L      X       X        Q        Hold
;       L      C      L       D        D        Load
;       L      C      H       X    Q PLUS 1     Increment
;      ------------------------------------------------

CHIP 9BitCoun   PAL20X10

CLK  D0 D1 D2 D3 D4 D5 D6 D7 D8 /LD GND
/OC /CO Q8 Q7 Q6 Q5 Q4 Q3 Q2 Q1  Q0 VCC

EQUATIONS

/Q0 :=  /LD*/Q0                                          ;Hold Q0
     +   LD*/D0                                          ;Load D0 (LSB)
    :+: /LD                                              ;Count

/Q1 :=  /LD*/Q1                                          ;Hold Q1
     +   LD*/D1                                          ;Load D1
    :+: /LD* Q0                                          ;Count

/Q2 :=  /LD*/Q2                                          ;Hold Q2
     +   LD*/D2                                          ;Load D2
    :+: /LD* Q0* Q1                                      ;Count

/Q3 :=  /LD*/Q3                                          ;Hold Q3
     +   LD*/D3                                          ;Load D3
    :+: /LD* Q0* Q1* Q2                                  ;Count

/Q4 :=  /LD*/Q4                                          ;Hold Q4
     +   LD*/D4                                          ;Load D4
    :+: /LD* Q0* Q1* Q2* Q3                              ;Count

/Q5 :=  /LD*/Q5                                          ;Hold Q5
     +   LD*/D5                                          ;Load D5
    :+: /LD* Q0* Q1* Q2* Q3* Q4                          ;Count

/Q6 :=  /LD*/Q6                                          ;Hold Q6
     +   LD*/D6                                          ;Load D6
    :+: /LD* Q0* Q1* Q2* Q3* Q4* Q5                      ;Count

/Q7 :=  /LD*/Q7                                          ;Hold Q7
     +   LD*/D7                                          ;Load D7
    :+: /LD* Q0* Q1* Q2* Q3* Q4* Q5* Q6                  ;Count

/Q8 :=  /LD*/Q8                                          ;Hold Q8
     +   LD*/D8                                          ;Load D8 (MSB)
    :+: /LD* Q0* Q1* Q2* Q3* Q4* Q5* Q6* Q7              ;Count

 CO :=  /LD*/Q0* Q1* Q2* Q3* Q4* Q5* Q6* Q7* Q8          ;Carry out
                                                         ;(Anticipate Count)
     +   LD* D0* D1* D2* D3* D4* D5* D6* D7* D8          ;Carry out
                                                         ;(Anticipate Load)
```

Figure 15. Design File for a Nine-bit Counter

```
SIMULATION

TRACE_ON /OC /LD D8 D7 D6 D5 D4 D3 D2 D1 D0
         /CO Q8 Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0

SETF OC LD /D8 /D7 /D6 /D5 /D4 /D3 /D2 /D1 /D0     ;Load
CLOCKF CLK

SETF /LD                                           ;Increment
CLOCKF CLK

SETF LD /D8 /D7 /D6 /D5 /D4 /D3 /D2 /D1 D0         ;Load
CLOCKF CLK

SETF /LD                                           ;Increment
CLOCKF CLK

SETF LD /D8 /D7 /D6 /D5 /D4 /D3 D2 D1 D0           ;Load
CLOCKF CLK

SETF /LD                                           ;Increment
CLOCKF CLK

SETF LD /D8 /D7 /D6 /D5 /D4 D3 D2 D1 D0            ;Load
CLOCKF CLK

SETF /LD                                           ;Increment
CLOCKF CLK

SETF LD /D8 /D7 /D6 /D5 D4 D3 D2 D1 D0             ;Load
CLOCKF CLK

SETF /LD                                           ;Increment
CLOCKF CLK

SETF LD /D8  D7  D6  D5 D4 D3 D2 D1 D0             ;Load
CLOCKF CLK


SETF /LD                                           ;Increment
CLOCKF CLK

TRACE_OFF
```

Figure 15. Design File for a Nine-bit Counter (Cont'd.)

Flip-flop Selection

Until now, all of the designs have been implemented in devices with D-type flip-flops. What happens if the counter design is implemented in a device which allows both J-K and T-type registers? The Boolean logic equations for such a design can be derived from the truth table. This requires advance knowledge of the functionality of the J-K and T-type registers. For the J-K register the output is asserted when the J input goes HIGH, and the output is unasserted when the K input goes HIGH. Toggle type registers require the T input to be asserted for every change in the output level.

PRESENT STATE				NEXT STATE															
				X3				X2				X1				X0			
X3	X2	X1	X0	D	J	K	T	D	J	K	T	D	J	K	T	D	J	K	T
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1
0	0	1	1	0	0	0	0	1	1	0	1	0	0	1	1	0	0	1	1
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1
0	1	0	1	0	0	0	0	1	0	0	0	1	1	0	1	0	0	1	1
0	1	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1
0	1	1	1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
1	0	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1
1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1
1	0	1	1	1	0	0	0	1	1	0	1	0	0	1	1	0	0	1	1
1	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0	1	0	0	0	1	1	0	1	0	0	1	1
1	1	1	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1
1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1

Figure 16. Truth Table for D, J-K and T-type Flip-flops

Figure 16 shows the truth table for both a J-K and a T-type register implementation for a binary counter. Deriving and optimizing the equations from the table, we get the following results:

```
X3-J := /X3 * X2 * X1 * X0
X3-K := X3 * X2 * X1 * X0

X2-J := /X2 * X1 * X0
X2-K := X2 * X1 * X0

X1-J := /X1 * X0
X1-K := X1 * X0

X0-J := /X0
X0-K := X0

X3-T := X2 * X1 * X0
X2-T := X1 * X0
X1-T := X0
X0-T := 1
```

As we can see from these equations, the number of product terms used for J-K and T-type implementations is smaller than the number of product terms required for a D-type implementation (Figure 12).

Which flip-flop is most efficient depends on the relative number of transitions or holds required. As a counter traverses from one count (state) to another, every output either makes a "transition" (changes logic level) or "holds" (stays at the same logic level). Small counters in general require more transitions and fewer holds. As the designs get larger, the higher-order bits require fewer transitions and more holds.

D-type flip-flops use up product terms only for active transitions from logic LOW level to HIGH level, and for logic HIGH level holds only. J-K and T-type flip-flops use up product terms for both LOW-to-HIGH and HIGH-to-LOW transitions, but eliminate hold terms. Generally, the requirements of transition and hold terms depends upon the count sequence selection. D-type flip-flops are more efficient for small designs. Conversely J-K and T-type flip-flops can be more efficient for large designs, which require more hold terms.

A comparison of product term requirements of 2-, 3-, 4- and 5-bit binary counters can be representative for other types of counters and state machines. The table in Figure 17 shows the transition terms and the hold terms required for these counters. For a J-K type flip-flop implementation, after optimizing, total product terms required are 4, 6, 8, and 10 respectively. The D-type implementation requires 3, 6, 10, and 15 respectively, and is relatively less efficient for large counters.

BINARY COUNTER	TRANSITIONS	HOLDS	D PRODUCT TERMS	J-K PRODUCT TERMS	T PRODUCT TERMS
2-Bit	6	2	3	4	1
3-Bit	14	10	6	6	1
4-Bit	30	34	10	8	1
5-Bit	62	98	15	10	1

Figure 17. Product Term Requirements for Configurable Flip-flops

There are a number of PLDs which offer the capability of programmable D, J-K and T-type registers. Figure 18 shows a list of all PLDs which offer configurable flip-flops. The J-K and T-type flip-flops are constructed from a basic D-type flip-flop using an extra XOR gate per flip-flop.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
22RX8	24	14	0-8	8-0	8	28.5
32VX10	24	12	0-10	10-0	8-16	25, 22.2

Figure 18. Registered PLDs with Configurable Flip-flops

Modulo Counters

The number of unique states a counter traverses is generally referred to as the modulus. A typical n-bit binary counter has a maximum modulus of 2^n. It is often necessary to introduce signal delays into the logic design to meet timing requirements. This makes it possible to allow for bus-skew, access time or differential propagation delays between devices along two different signal paths. A typical example of this is the introduction of wait states to allow for access times of different memory elements. Counters and delay lines are commonly used to introduce the delay. Counters in PLDs have the added advantage of programmability to select the required delay. Such applications where precise timing duration control is required usually use modulo counters with a non-power-of-two modulus. Other applications of modulo counters include waveform generators and arbiters.

A good example of a modulo counter is a BCD counter. Such a counter is useful in applications where the computer's outputs are generated using a decimal system. While a four-bit binary counter can count up to sixteen, the BCD counter terminates the count at the modulus of 10.

Modulo counters can be designed in a variety of ways. One direct way is to use the truth table to implement a count to a modulus and directly derive the equations from it. The truth table for a BCD count (from zero to nine) is shown in Figure 19.

Now let us consider what happens if the device accidentally powers up in one of the count values from ten to fifteen. These are illegal counts (states) and, for a good design, a mechanism must be built into the equations to allow it to recover back into a legal state. What we actually need is to consider the truth table in Figure 20 in conjunction with the one in Figure 19 for deriving the Boolean equations.

PRESENT STATE					NEXT STATE			
Q3	Q2	Q1	Q0		Q3	Q2	Q1	Q0
0	0	0	0	0 -> 1	0	0	0	1
0	0	0	1	1 -> 2	0	0	1	0
0	0	1	0	2 -> 3	0	0	1	1
0	0	1	1	3 -> 4	0	1	0	0
0	1	0	0	4 -> 5	0	1	0	1
0	1	0	1	5 -> 6	0	1	1	0
0	1	1	0	6 -> 7	0	1	1	1
0	1	1	1	7 -> 8	1	0	0	0
1	0	0	0	8 -> 9	1	0	0	1
1	0	0	1	9 -> 0	0	0	0	0

Figure 19. Truth Table for a BCD Counter

PRESENT STATE					NEXT STATE			
Q3	Q2	Q1	Q0		Q3	Q2	Q1	Q0
1	0	1	0	10 -> 0	0	0	0	0
1	0	1	1	11 -> 0	0	0	0	0
1	1	0	0	12 -> 0	0	0	0	0
1	1	0	1	13 -> 0	0	0	0	0
1	1	1	0	14 -> 0	0	0	0	0
1	1	1	1	15 -> 0	0	0	0	0

Figure 20. Truth Table for Illegal State Recovery to Count Zero

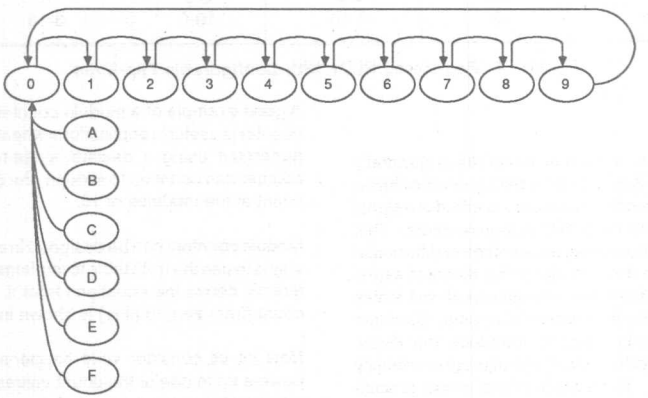
or the PALASM 2 software minimizer.

The Boolean equation for Q3 is:

```
/Q3 :=  /Q3 * /Q2 * /Q1 * /Q0
      + /Q3 * /Q2 * /Q1 *  Q0
      + /Q3 * /Q2 *  Q1 * /Q0
      + /Q3 * /Q2 *  Q1 *  Q0
      + /Q3 *  Q2 * /Q1 * /Q0
      + /Q3 *  Q2 * /Q1 *  Q0
      + /Q3 *  Q2 *  Q1 * /Q0
      +  Q3 * /Q2 * /Q1 *  Q0
      +  Q3 * /Q2 *  Q1 * /Q0
      +  Q3 * /Q2 *  Q1 *  Q0
      +  Q3 *  Q2 * /Q1 * /Q0
      +  Q3 *  Q2 * /Q1 *  Q0
      +  Q3 *  Q2 *  Q1 * /Q0
      +  Q3 *  Q2 *  Q1 *  Q0
```

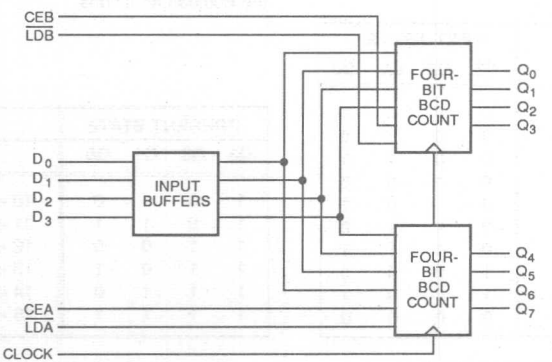
```
      + /Q3 * /Q1
      + /Q2 *  Q0
      +  Q3 *  Q1
      +  Q3 *  Q2
```

Similar Boolean equations can be generated for Q2, Q1 and Q0. Figure 22 shows the circuit diagram of a loadable dual BCD counter implemented in a PAL16R8. Figure 23 shows the design file for such a counter.



406 07

Figure 21. State Sequence of a BCD Counter Showing Illegal State Recovery



406 08

Figure 22. Circuit of a Dual BCD Counter

```
TITLE           DUAL BCD COUNTER.
PATTERN         00
REVISION        01.
AUTHOR          CHRIS JAY.
COMPANY         MMI SANTA CLARA, CA.
DATE            17 JULY 1987
;
CHIP BCD PAL16R8
;
;THE PAL16R8 HAS BEEN PROGRAMMED AS A DUAL BINARY
;CODED DECIMAL COUNTER. REGISTERS MAY BE LOADED FROM
;DATA INPUTS D0 - D3, THE LDA ENABLES DATA ON
;D0 - D3 TO BE SYNCHRONOUSLY LOADED INTO Q4 - Q7.
;LDB ENABLES DATA ON D0 - D3 TO BE LOADED INTO
;REGISTERS Q0 - Q3. TWO COUNT ENABLE INPUTS CEA
;AND CEB ENABLE A BINARY CODED DECIMAL COUNT IN
;BOTH COUNTERS. THE COUNTERS RESET IF POWERED UP
;IN A HEXADECIMAL STATE, SO IF USED IN A STAND
;ALONE MODE THE COUNTERS WILL NOT LOCK UP INTO
;AN INVALID STATE.
;
;PIN    1       2       3       4       5
        CLK     /LDA    CEA     D0      D1

;PIN    6       7       8       9       10
        D2      D3      CEB     /LDB    GND

;PIN    11      12      13      14      15
        /OE     Q0      Q1      Q2      Q3

;PIN    16      17      18      19      20
        Q4      Q5      Q6      Q7      VCC

                                        ;
STRING  RESA    'Q5*Q7 + Q6*Q7'         ;RESET FROM HEX
STRING  RESB    'Q1*Q3 + Q2*Q3'         ;STATES
;
EQUATIONS                               ;
                                        ;
/Q0     :=      Q0*/LDB*CEB             ;COUNT EQUATION
         +      RESB                    ;RESET FROM ILLEGAL STATE
         +      /D0*LDB*/CEB            ;LOAD D0
         +      /Q0*/CEB*/LDB           ;HOLD Q0
                                        ;
/Q1     :=      Q3*/LDB*CEB             ;COUNT EQUATION
         +      Q0*Q1*/LDB*CEB          ;
         +      /Q0*/ Q1*/LDB*CEB       ;
         +      RESB                    ;RESET FROM ILLEGAL STATE
         +      /D1*LDB*/CEB            ;LOAD D1
         +      /Q1*/CEB*/LDB           ;HOLD Q1
                                        ;
/Q2     :=      Q3*/LDB*CEB             ;COUNT EQUATION
         +      /Q0*/Q2*/LDB*CEB        ;
         +      /Q1*/Q2*/LDB*CEB                ;
         +      Q0* Q1*Q2*/LDB*CEB      ;
         +      RESB                    ;RESET FROM ILLEGAL STATE
         +      /D2*LDB*/CEB            ;LOAD D2
         +      /Q2*/CEB*/LDB           ;HOLD Q2
                                        ;
/Q3     :=      /Q3*/Q0*/LDB*CEB        ;COUNT EQUATION
         +      Q0*/Q2*/LDB*CEB         ;
         +      /Q1*/Q3*/LDB*CEB        ;
         +      RESB                    ;RESET FROM ILLEGAL STATE
         +      /D3*LDB*/CEB            ;LOAD D3
         +      /Q3*/CEB*/LDB           ;HOLD Q3
                                        ;
/Q4     :=      Q4*/LDA*CEA             ;COUNT EQUATION
         +      RESA                    ;RESET FROM ILLEGAL STATE
         +      /D0*LDA*/CEA            ;LOAD D0
         +      /Q4*/CEA*/LDA           ;HOLD Q4
                                        ;
```

Figure 23. Design File for a Dual BCD Counter

```
/Q5     :=      Q7*/LDA*CEA             ;COUNT EQUATION
         +      Q4*Q5*/LDA*CEA          ;
         +      /Q4*/Q5*/LDA*CEA        ;
         +      RESA                    ;RESET FROM ILLEGAL STATE
         +      /D1*LDA*/CEA            ;LOAD D1
         +      /Q5*/CEA*/LDA           ;HOLD Q5
                                        ;
/Q6     :=      Q7*/LDA*CEA             ;COUNT EQUATION
         +      /Q4*/Q6*/LDA*CEA        ;
         +      /Q5*/Q6*/LDA*CEA        ;
         +      Q4*Q5*Q6*/LDA*CEA       ;
         +      RESA                    ;RESET FROM ILLEGAL STATE
         +      /D2*LDA*/CEA            ;LOAD D2
         +      /Q6*/CEA*/LDA           ;HOLD Q6
                                        ;
/Q7     :=      /Q7*/Q4*/LDA*CEA        ;COUNT EQUATION
         +      Q4*/Q6*/LDA*CEA         ;
         +      /Q5*/Q7*/LDA*CEA        ;
         +      RESA                    ;RESET FROM ILLEGAL STATE
         +      /D3*LDA*/CEA            ;LOAD D3
         +      /Q7*/CEA*/LDA           ;HOLD Q7
                                        ;

SIMULATION                              ;
TRACE_ON        CLK Q4 Q5 Q6 Q7         ;TRACE ALL SIGNALS
                Q0 Q1 Q2 Q3 LDA         ;
                LDB D0 D1 D2 D3         ;
                CEA CEB  OE             ;
;PRLDF  Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7         ;
SETF    /CLK OE                         ;
CLOCKF  CLK                             ;
SETF    /CLK OE D0 D1 D2 /D3            ;SET DATA INPUTS SELECT
SETF    LDA LDB /CEA /CEB               ;REGISTERS Q0 - Q3 PERFORM
CLOCKF  CLK                             ;SYNCHRONOUS LOAD.
SETF    /LDB /LDA                       ;SELECT REGISTERS Q4 - Q7
CLOCKF  CLK                             ;PERFORM SYNCHRONOUS LOAD.
SETF    CEA /LDA                        ;ENABLE Q4 - Q7 AS A BCD
FOR I := 1 TO 10 DO                     ;COUNTER, CLOCK FOR 10
BEGIN CLOCKF CLK                        ;CLOCK CYCLES
END                                     ;
SETF /CEA CEB /LDB                      ;ENABLE Q0 - Q3 AS A BCD
FOR I := 1 TO 10 DO                     ;COUNTER, CLOCK FOR 10
BEGIN CLOCKF CLK                        ;CYCLES
END                                     ;
SETF /CEB LDB /D0 D1 /D2 D3             ;LOAD ILLEGAL STATES
CLOCKF CLK                              ;TO TEST COUNTER RESET
SETF /LDB                               ;
CLOCKF CLK                              ;
SETF LDB D0 D1 /D2 D3                   ;
CLOCKF CLK                              ;
SETF /LDB                               ;
CLOCKF CLK                              ;
SETF LDB /D0 /D1 D2 D3                  ;
CLOCKF CLK                              ;
SETF /LDB                               ;
CLOCKF CLK                              ;
SETF LDB D0 /D1 D2 D3                   ;
CLOCKF CLK                              ;
SETF /LDB                               ;
CLOCKF CLK                              ;
SETF LDB /D0 D1 D2 D3                   ;
CLOCKF CLK                              ;
SETF /LDB                               ;
CLOCKF CLK                              ;
SETF LDB D0 D1 D2 D3                    ;
CLOCKF CLK                              ;
SETF /LDB                               ;
CLOCKF CLK                              ;
TRACE_OFF                               ;
```

Figure 23. Design File for a Dual BCD Counter (Cont'd.)

Modulo Counter Device Selection Considerations

We have illustrated a counter which counts from zero to a fixed modulus. The same technique can be applied for a counter which counts down from a maximum power-of-two number to a fixed modulus, or even a counter which counts from one modulus to another. The important considerations will be the number of product terms used.

The registered PLDs used for modulo counters are similar to the ones selected for other counters. Since the counts used are binary, devices with J-K or T-type flip-flops or XOR gates will help optimize the number of product terms used. The product term usage also depends upon the modulus selected. Generally a power-of-two or a multiple-of-two modulus will require fewer product terms.

Another factor for flip-flop selection is the illegal states. D-type flip-flops are generally better suited for illegal state recovery than the J-K or T-type flip-flops. This is because when no product term is asserted, the D-type flip-flops reset to zero. Designers using J-K or T-type flip-flops must design-in illegal state recovery.

Certain devices allow the use of a synchronous RESET product term for modulo counters. The idea is to use a minimal number of product terms to build a binary counter which counts up to a power-of-two number. However, this counter is RESET to zero using the synchronous RESET product term when the desired modulus is reached. It then begins counting afresh from zero, and the procedure is repeated. Similar operation can also be achieved with a synchronous PRESET product term for a down counter. Figure 24 shows a table of all the PLDs that provide the RESET and PRESET product terms.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16RA8	20	8	0-8	8-0	4	20
23S8	20	9	4	4	8-12	33.3, 28.5
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
CMOS						
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
10H20EV/EG8	24	12	0-8	8-0	8-12	125
10020EV/EG8	24	12	0-8	8-0	8-12	125

Figure 24. PLDs Which Provide RESET or PRESET Product Terms

Using synchronous RESET and PRESET product terms allows the counter to recover from illegal states. Notice that the logic product terms in the counter are designed for a complete binary count. If the counter powers up in any illegal state (as shown in Figure 25), it will continue the count until the terminal count and return to zero, where the correct modulo count will begin. This illegal state recovery will take an unpredictable number of clock cycles, and you may wish to design a more systematic recovery system.

Cascading Modulo Counters

For large modulo counters, the technique of generating Boolean equations from the truth tables is very tedious and time consuming. Another approach for designing modulo counters is to divide it into two smaller modulo counters. In addition to simplifying the design, this approach usually helps optimize the number of product terms.

As an example, a modulo-360 counter can be directly implemented with nine register bits. However, instead of implementing this as a straight 9-bit counter, we can implement this as two counters: one four-bit counter (counting from zero to 14) and another five-bit counter (counting from zero to 23). Together the two counters count up to 360. Shown in Figure 26 the terminal count output MOUT is asserted when the count reaches 360.

The design requires nine inputs, nine outputs, one clock pin, one LOAD pin, one RESET and one MOUT (module output signal) pin. Note that no extra flip-flops or pins were needed. Obviously, the count values of this counter are not the same as a straight modulo-360 counter. Actually, this is what contributes to the optimization of the number of product terms used. The PLPL design file of this counter is shown in Figure 27.

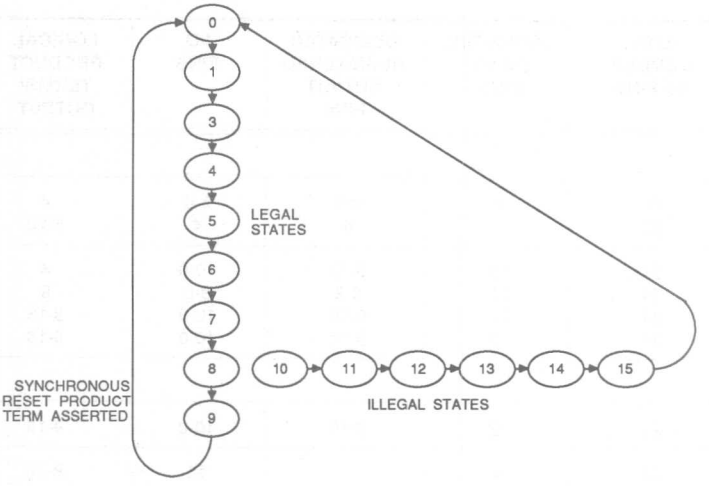


Figure 25. A BCD Counter Using Synchronous RESET Product Term

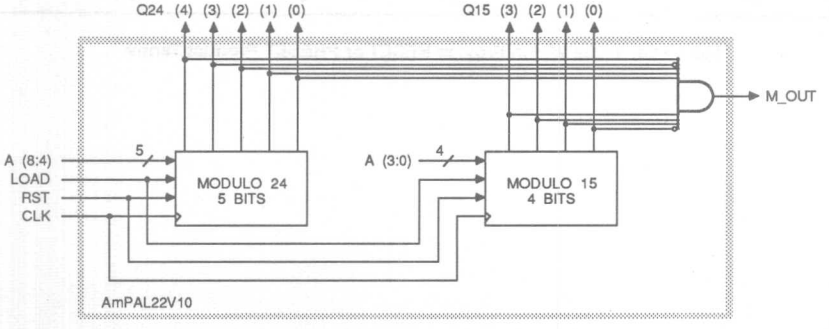


Figure 26. A Modulo-360 Counter in PAL22V10

```
DEVICE  MODULO_360_COUNTER  (PAL22V10)

PIN       CLK   = 1   RST  = 2
          LOAD  = 3
          A[8:0] = 4:11,13
          Q24[4:0] = 18:14
          Q15[3:0] = 22:19
          M_OUT = 23; "ASSERTED WHEN COUNT REACHES 360"

BEGIN
     IF (RST)  THEN ARESET() ;    "COUNTER INITIALIZATION"
     IF (LOAD) THEN BEGIN
           Q24[4:0] := A[8:4] ;  "INITIALIZATION COUNT"
           Q15[3:0] := A[3:0] ;  "INITIALIZATION COUNT"
     END;
     ELSE BEGIN
           IF (Q24[4]*/Q24[3]* Q24[2]* Q24[1]* Q24[0]*
              Q15[3]* Q15[2]* Q15[1]*/Q15[0]) THEN M_OUT = 1;
           CASE (Q24[4:0]) BEGIN "MODULO 24"
                 #B00000) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B00001 ;
                 END ;

                 #B00001) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B00010 ;
                 END ;

                 #B00010) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B00011 ;
                 END ;

                 #B00011) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B00100 ;
                 END ;

                 #B00100) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B00101 ;
                 END ;

                 #B00101) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B00110 ;
                 END ;

                 #B00110) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B00111 ;
                 END ;
                 #B00111) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B01000 ;
                 END ;

                 #B01000) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B01001 ;
                 END ;

                 #B01001) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B01010 ;
                 END ;

                 #B01010) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B01011 ;
                 END ;

                 #B01011) BEGIN
                          Q15[3:0] := Q15[3:0] ;
                          Q24[4:0] := #B01100 ;
                 END ;
```

Figure 27. Design File of a Modulo-360 Counter

```
                    #B01100) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B01101 ;
                    END ;

                    #B01101) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B01110 ;
                    END ;

                    #B01110) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B01111 ;
                    END ;

                    #B01111) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10000 ;
                    END ;

                    #B10000) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10001 ;
                    END ;

                    #B10001) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10010 ;
                    END ;

                    #B10010) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10011 ;
                    END ;

                    #B10011) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10100 ;
                    END ;

                    #B10100) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10101 ;
                    END ;

                    #B10101) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10110 ;
                    END ;

                    #B10110) BEGIN
                             Q15[3:0] := Q15[3:0] ;
                             Q24[4:0] := #B10111 ;
                    END ;

                    #B10111) BEGIN
                             Q24[3:0] := #B00000 ;
                             CASE (Q15[3:0] BEGIN  "MODULO 15"
                                    #B0000) Q15[3:0] := #B0001 ;
                                    #B0001) Q15[3:0] := #B0010 ;
                                    #B0010) Q15[3:0] := #B0011 ;
                                    #B0011) Q15[3:0] := #B0100 ;
                                    #B0100) Q15[3:0] := #B0101 ;
                                    #B0101) Q15[3:0] := #B0110 ;
                                    #B0110) Q15[3:0] := #B0111 ;
                                    #B0111) Q15[3:0] := #B1000 ;
                                    #B1000) Q15[3:0] := #B1001 ;
                                    #B1001) Q15[3:0] := #B1010 ;
                                    #B1010) Q15[3:0] := #B1011 ;
                                    #B1011) Q15[3:0] := #B1100 ;
                                    #B1100) Q15[3:0] := #B1111 ;
                                    #B1101) Q15[3:0] := #B1110 ;
                                    #B1110) Q15[3:0] := #B0000 ;
                              END ;
                    END ;
               END ;
          END ;
END.
```

Figure 27. Design File of a Modulo-360 Counter (Cont'd.)

```
 TEST_VECTORS

 IN  CLK RST LOAD A[8:0] ;

OUT          Q24[4:0] Q15[3:0] M_OUT ;

BEGIN

"CLK RST LOAD       A[8:0]          Q24[4:0]   Q15[3:0]   M_OUT"
 X   1    X    X X X X X X X X X   L L L L L   L L L L   L;
 C   0    1    1 0 1 0 1 1 1 0 0   H L H L H   H H L L   L;
 C   0    0    X X X X X X X X X   H L H H L   H H L L   L;
 C   0    0    X X X X X X X X X   H L H H H   H H L L   L;
 C   0    0    X X X X X X X X X   L L L L L   H H L H   L;
 C   0    1    0 1 1 1 1 1 1 1 0   L H H H H   H H H L   L;
 C   0    0    X X X X X X X X X   H L L L L   H H H L   L;
 C   0    0    X X X X X X X X X   H L L L H   H H H L   L;
 C   0    0    X X X X X X X X X   H L L H L   H H H L   L;
 C   0    0    X X X X X X X X X   H L L H H   H H H L   L;
 C   0    0    X X X X X X X X X   H L H L L   H H H L   L;
 C   0    0    X X X X X X X X X   H L H L H   H H H L   L;
 C   0    0    X X X X X X X X X   H L H H L   H H H L   L;
 C   0    0    X X X X X X X X X   H L H H H   H H H L   H;
 C   0    0    X X X X X X X X X   L L L L L   L L L L   L;
 C   0    0    X X X X X X X X X   L L L L H   L L L L   L;
 C   0    0    X X X X X X X X X   L L L H L   L L L L   L;
END.
```

Figure 27. Design File of a Modulo-360 Counter (Cont'd.)

PLS Devices for Modulo Counters

Monolithic Memories offers a family of PLS devices. PLS devices provide programmable AND and programmable OR arrays. These devices offer S-R type output and buried flip-flops. They also provide a complement array term which is very useful for designing modulo counters. The available PLS devices are listed in Figure 28.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
PLS105	28	16	8	-	Total 48	37
PLS167	24	14	6	-	Total 48	33
PLS168	24	12	8	-	Total 48	33

Figure 28. PLS Devices

state recovery and programmable modulo counter designs.

Illegal State Recovery

For any counter going through a sequence of legal counts (states), at least one product term is always active as long as the counter never needs to hold a count; such terms determine the next legal state. When the counter powers up in any invalid state, none of the product terms for deciding the next state will be active. When none of the product terms connected to the complement array is active, the complement array is asserted.

The complement array is usually connected to all of the product terms of the counter. Thus when the count is legal, the complement array output is always LOW (Figure 30). When the counter is in an illegal state, the complement array detects it and is

Programmable Modulus Counter

The complement array term can instead be used for designing programmable modulus counters. A counter is implemented with the device registers. Different product terms are programmed with different modulus values. These product terms are under the control of input signals. At any one time only one of the product terms is kept active. The complement array is connected to all such terminal count product terms. Since all of these product terms are LOW the complement array is initially HIGH and is used as a count enable for the counter. When the count reaches the terminal value the complement array term goes LOW disabling the normal count product terms. Simultaneously the terminal count product term becomes active, resetting the counter to begin counting afresh.

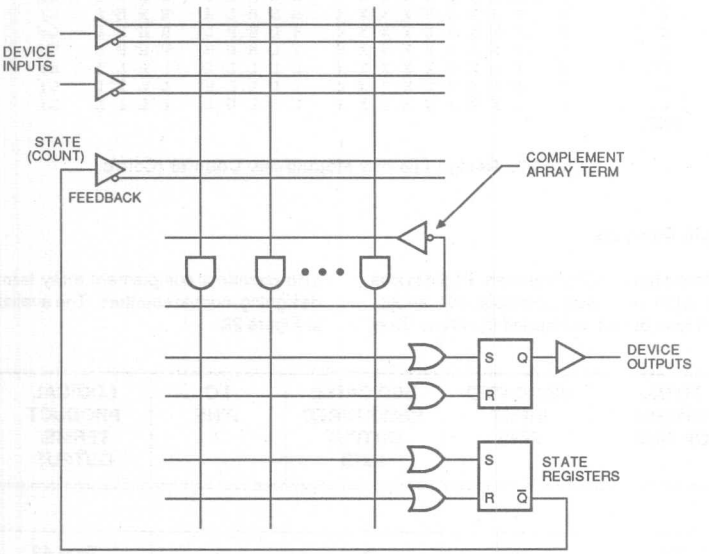


Figure 29. The Architecture of the Complement Array Term

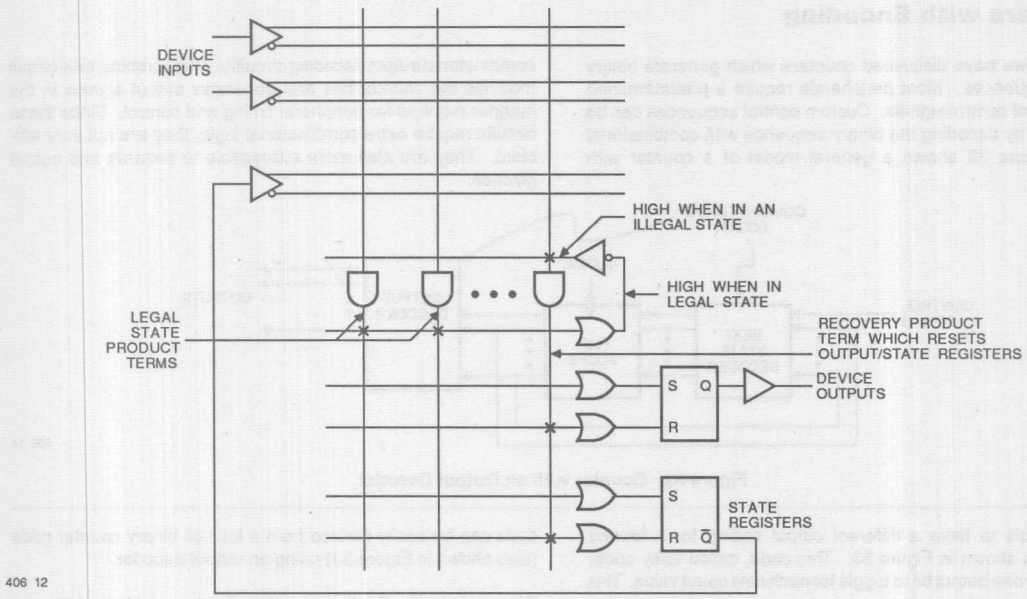


Figure 30. Complement Array for Illegal State Recovery

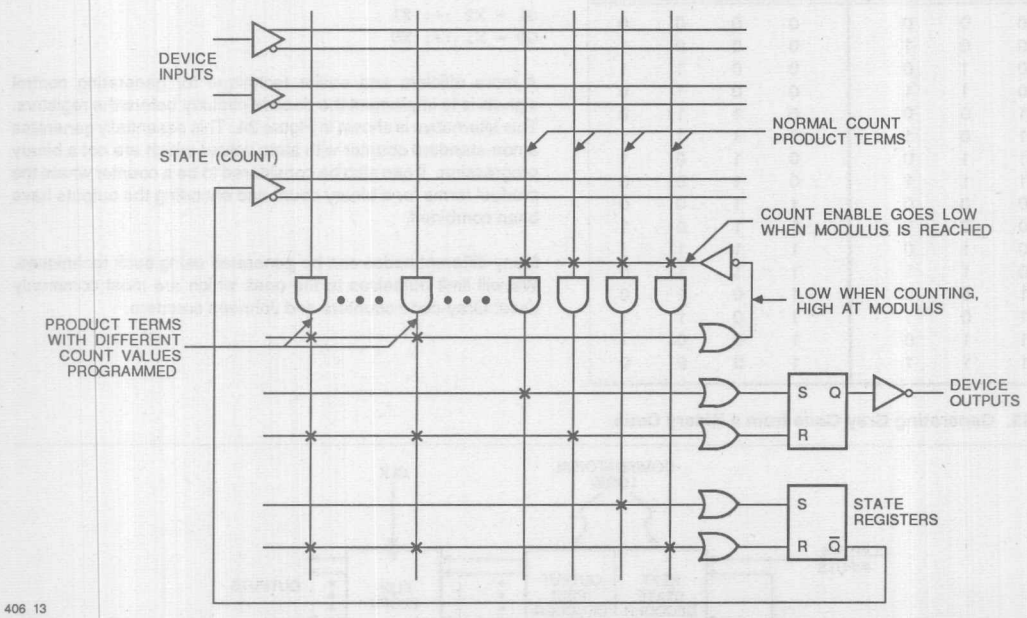


Figure 31. Programmable Modulo Counter Using Complement Array Term

Counters with Encoding

Until now we have discussed counters which generate binary output sequences. Most peripherals require a predetermined sequence of control signals. Custom control sequences can be generated by decoding the binary sequence with combinatorial logic. Figure 32 shows a general model of a counter with combinatorial output decoding circuitry. This combinatorial circuit modifies the counter bits and generates output signals in the manner required for peripheral timing and control. Since these circuits require extra combinatorial logic, they are not very efficient. They are also more susceptible to hazards and output glitches.

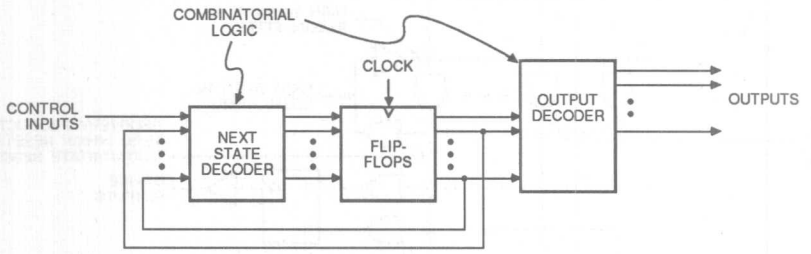


Figure 32. Counter with an Output Decoder

It is possible to have a different output coding for a four-bit counter, as shown in Figure 33. This code, called Gray code, allows only one output bit to toggle for each new count value. This code can be easily derived from a four-bit binary counter code (also shown in Figure 33) using an output decoder.

BINARY CODE				GRAY CODE			
X3	X2	X1	X0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Figure 33. Generating Gray Code from a Binary Code

We can derive the Boolean equations for the combinatorial output decoder from the truth table. The equations are:

```
G3 = X3
G2 = X3 :+: X2
G1 = X2 :+: X1
G0 = X1 :+: X0
```

A more efficient and easier technique for generating control signals is to implement the decode circuitry before the registers. This alternative is shown in Figure 34. This essentially generates a non-standard counter with state values which are not a binary progression. It can also be considered to be a counter where the product terms for a binary count and encoding the outputs have been combined.

Many different codes can be generated using such techniques. We will limit ourselves to the ones which are most commonly used: Gray-code counters and Johnson counters.

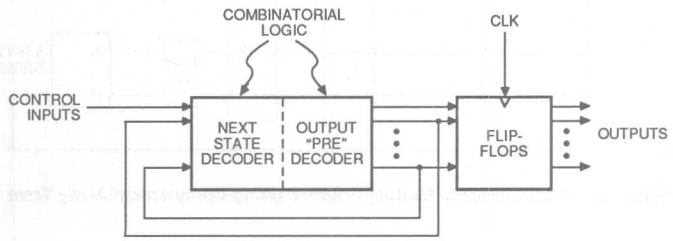


Figure 34. Counter with Combined Next State Generation and Output Encoding Circuit

Gray-Code Counters

Gray-code counters are often used in digital designs for control timing functions. The primary advantage of Gray-code counters stems from the characteristic that only one output bit changes value for every clock cycle. These output signals can then easily be decoded using a combinatorial decoder without any risk of hazards. Gray-code counters are used extensively as system clocks, since the different output bits provide different clock phases without the risks of hazards. Gray-code is also used in high-speed data communication applications, where data is transmitted from one part of the system to another, and where the error susceptibility increases with the number of bit changes between adjacent numbers in a sequence. These are also used for such specialized applications as shaft encoders and real-time process control.

The implementation of a Gray-code counter is very simple. A truth table can be derived from the transition table as is done for a binary counter. The Boolean equations can then be directly derived from the truth table. The truth table for the Gray-code counter is shown in Figure 35.

PRESENT STATE				NEXT STATE			
X3	X2	X1	X0	X3	X2	X1	X0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	0	1	0	1
0	1	0	1	0	1	0	0
0	1	0	0	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	0
1	1	1	0	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	0	0	1
1	0	0	1	1	0	0	0
1	0	0	0	0	0	0	0

Figure 35. Truth Table for a Four-bit Gray-code Counter

The Boolean logic equations for a Gray-code counter are:

```
X3 := /X3 *  X2 * /X1 * /X0
       X3 *  X2 * /X1 * /X0
       X3 *  X2 * /X1 *  X0
       X3 *  X2 *  X1 *  X0
       X3 *  X2 *  X1 * /X0
       X3 * /X2 *  X1 * /X0
       X3 * /X2 *  X1 *  X0
       X3 * /X2 * /X1 *  X0

X2 := /X3 * /X2 *  X1 * /X0
      /X3 *  X2 *  X1 * /X0
      /X3 *  X2 *  X1 *  X0
      /X3 *  X2 * /X1 *  X0
      /X3 *  X2 * /X1 * /X0
       X3 *  X2 * /X1 * /X0
       X3 *  X2 * /X1 *  X0
       X3 *  X2 *  X1 *  X0

X1 := /X3 * /X2 * /X1 *  X0
      /X3 * /X2 *  X1 *  X0
      /X3 * /X2 *  X1 * /X0
      /X3 *  X2 *  X1 * /X0
       X3 *  X2 * /X1 *  X0
       X3 *  X2 *  X1 *  X0
       X3 *  X2 *  X1 * /X0
       X3 * /X2 *  X1 * /X0

X0 := /X3 * /X2 * /X1 * /X0
      /X3 * /X2 * /X1 *  X0
      /X3 *  X2 *  X1 * /X0
      /X3 *  X2 *  X1 *  X0
       X3 *  X2 * /X1 * /X0
       X3 *  X2 * /X1 *  X0
       X3 * /X2 *  X1 * /X0
       X3 * /X2 *  X1 *  X0
```

A Johnson counter is part of a family of counters known as "ring counters." These counters are used for special applications where code symmetry is desired. Ring counters are also often used for timing purposes, since all of the outputs are essentially a series of pulses. This code symmetry also allows use of the fewest possible product terms with a D-type register. Devices such as the Logic Cell Array (LCA), which provide a small amount of logic per cell, can implement Johnson counters very easily.

Johnson counters are also known as circular-shift counters. The sequence for a five-stage Johnson counter is shown in Figure 36. As can be seen in the truth table, the counter first fills up with 1's from left to right and then it fills up with zeros again. Note from the output sequence that only one of the Johnson counter bits changes for every clock period, like the Gray-code counter. One major advantage of the Johnson counter is that it can be readily decoded with small two-input NAND gates and hence is suitable for high-speed applications.

Note that the five-stage sequence has a table of 10 legal states and 22 illegal states (Figure 37). In general, an n-bit Johnson counter will produce a modulus of 2n. Figure 38 shows the state diagram of the five-bit counter.

Legal States

PRESENT STATE					NEXT STATE				
Q4	Q3	Q2	Q1	Q0	Q4	Q3	Q2	Q1	Q0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	1	1	0	0	0	0	1
0	0	1	1	1	0	0	0	1	1
0	1	1	1	1	0	0	1	1	1
1	1	1	1	1	0	1	1	1	1
1	1	1	1	0	1	1	1	1	1
1	1	1	0	0	1	1	1	1	0
1	1	0	0	0	1	1	1	0	0
1	0	0	0	0	1	1	0	0	0

Figure 36. Five-bit Johnson Counter Truth Table

PRESENT STATE					NEXT STATE				
Q4	Q3	Q2	Q1	Q0	Q4	Q3	Q2	Q1	Q0
0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0

Figure 37. Illegal States for a Five-bit Johnson Counter

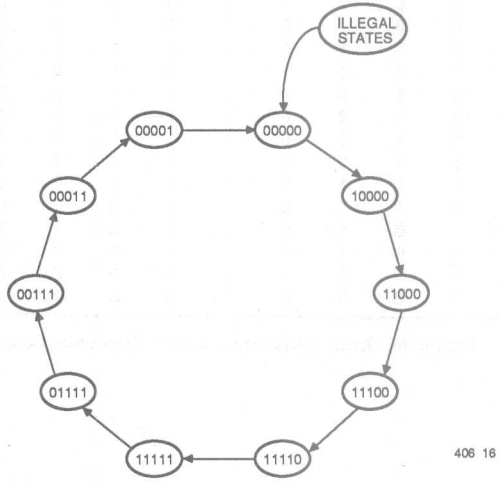


Figure 38. State Diagram of a Five-bit Johnson Counter

The implementation of a Johnson counter is relatively straightforward, and is the same regardless of the number of stages. When D-type flip-flops are used, the Q output of each flip-flop is connected to the D input of the following stage. The single exception is the Q output of the last stage, which is complemented and connected to the D input of the first stage.

One disadvantage of the counter is the number of invalid (or illegal) states. The invalid states increase exponentially with the length of the counter. The bigger the counter becomes, the greater are its chances of entering an illegal state. Johnson counters are very susceptible to illegal states, and can "hang up" very easily. Noise or improper use can cause this counter to end up in an illegal state. Therefore a design with illegal state recovery circuitry is always recommended.

Figure 39 shows a nine-bit Johnson counter which can be derived by directly extending the design of a five-bit Johnson counter. The design file adds a LOAD function; it requires nine input pins, nine output pins, one RESET pin, one LOAD pin, one invalid status pin and one clock pin. The design incorporates the product terms required for illegal state recovery. The PLPL design file for this counter is shown in Figure 40.

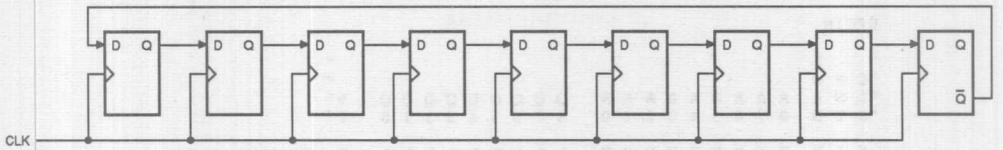


Figure 39. Block Diagram of a Nine-bit Johnson Counter

```
DEVICE   NINE_BIT_JOHNSON_COUNTER (AmPAL22V10)


PIN CLK   = 1
    RST   = 2
    A[8:0] = 3:11
    Q[8:0] = 14:21,23
    VF    = 22 "VALID STATE FLAG"
    LOAD = 13 ;


BEGIN

    IF (RST)  THEN ARESET(Q[8:0]) ;     "COUNTER INITIALIZATION"
    IF (LOAD) THEN Q[8:0] := A[8:0] ;  "LOAD NEW VALUE"
    ELSE BEGIN
           CASE (Q[8:0])
           BEGIN
                #B000000000)  Q[8:0] := #B100000000 ;
                #B100000000)  Q[8:0] := #B110000000 ;
                #B110000000)  Q[8:0] := #B111000000 ;
                #B111000000)  Q[8:0] := #B111100000 ;
                #B111100000)  Q[8:0] := #B111110000 ;
                #B111110000)  Q[8:0] := #B111111000 ;
                #B111111000)  Q[8:0] := #B111111100 ;
                #B111111100)  Q[8:0] := #B111111110 ;
                #B111111110)  Q[8:0] := #B111111111 ;
                #B111111111)  Q[8:0] := #B011111111 ;
                #B011111111)  Q[8:0] := #B001111111 ;
                #B001111111)  Q[8:0] := #B000111111 ;
                #B000111111)  Q[8:0] := #B000011111 ;
                #B000011111)  Q[8:0] := #B000001111 ;
                #B000001111)  Q[8:0] := #B000000111 ;
                #B000000111)  Q[8:0] := #B000000011 ;
                #B000000011)  Q[8:0] := #B000000001 ;
                #B000000001)  Q[8:0] := #B000000000 ;
           END ;
         END ;
    CASE (Q[8:0])
         BEGIN
                #B000000000)  VF = 1 ;
                #B100000000)  VF = 1 ;
                #B110000000)  VF = 1 ;
                #B111000000)  VF = 1 ;
                #B111100000)  VF = 1 ;
                #B111110000)  VF = 1 ;
                #B111111000)  VF = 1 ;
                #B111111100)  VF = 1 ;
                #B111111110)  VF = 1 ;
                #B111111111)  VF = 1 ;
                #B011111111)  VF = 1 ;
                #B001111111)  VF = 1 ;
                #B000111111)  VF = 1 ;
                #B000011111)  VF = 1 ;
                #B000001111)  VF = 1 ;
                #B000000111)  VF = 1 ;
                #B000000011)  VF = 1 ;
                #B000000001)  VF = 1 ;
         END ;

END.
```

Figure 40. Design File for a Nine-bit Johnson Counter

```
TEST_VECTORS

IN    CLK RST LOAD A[8:0];
OUT   Q[8:0] VF;

BEGIN

"    L                                                    "
"C R O                                                    "
"L S A  A A A A A A A A A   Q Q Q Q Q Q Q Q Q   V"
"K T D  8 7 6 5 4 3 2 1 0   8 7 6 5 4 3 2 1 0   F"

 X 1 X  X X X X X X X X X   L L L L L L L L L   H; "RESET"
 C 0 1  1 1 1 1 1 1 1 1 1   H H H H H H H H H   H; "LOAD"
 C 0 0  X X X X X X X X X   L H H H H H H H H   H; "COUNT"
 C 0 0  X X X X X X X X X   L L H H H H H H H   H;
 C 0 0  X X X X X X X X X   L L L H H H H H H   H; "COUNT"
 C 0 1  1 0 1 0 1 0 1 0 1   H L H L H L H L H   L; "LD INVALID ST"
 C 0 0  X X X X X X X X X   L L L L L L L L L   H; "INVALID ST"
 C 0 0  X X X X X X X X X   H L L L L L L L L   H; "RECOVERY"
 C 0 1  1 1 1 1 1 1 1 1 0   H H H H H H H H L   H; "LOAD"
 C 0 0  X X X X X X X X X   H H H H H H H H H   H; "COUNT"
 C 0 0  X X X X X X X X X   L H H H H H H H H   H;
 C 0 1  0 0 0 0 0 0 0 0 1   L L L L L L L L H   H;
 C 0 0  X X X X X X X X X   L L L L L L L L L   H;
END.
```

Figure 40. Design File for a Nine-bit Johnson Counter (Cont'd.)

Shift Registers

A Shift Register is a special digital circuit often used as a primary building block in digital computer systems. It is closely related to a ring counter. Its fundamental usage is for temporary data storage and bit-wise data manipulation for advanced arithmetic and multiplication operations. Shift registers are also frequently used in communications, for converting parallel byte-wide data from the microprocessor to a serial data bit-stream for transmission. An example of a serial data link controller on page 2-453 shows such a shift register design. Shift registers are also used in graphics systems for serializing parallel data for use by the display monitor. A number of examples of video shift registers are included in the graphics section (page 2-257).

The fundamental purpose of a shift register (Figure 41) is to shift data from one flip-flop to another. There are several types of shift registers. They are classified by the way in which incoming data is received (parallel or serial), and how outgoing data is transmitted (parallel or serial).

In the following example, we will discuss a simple universal shifter which provides both serial and parallel input and output functions. Depending upon the control signals I0 and I1, the data is shifted from one flip-flop to another in the left or the right direction. These inputs also control when the new parallel data is loaded onto the registers. When shifting left or right, serial data can be received and transmitted on serial pins LIRO and RILO. Since the flip-flop outputs appear on the output pins at all times, the parallel output data is always available. The truth table is shown in Figure 42.

The Boolean logic equations can be directly derived from the truth table, and are shown in the PAL device design file in Figure 43.

Shift registers can be modified to suit various system design requirements. This universal shift register can be used for serial in/serial out, parallel in/parallel out, serial in/parallel out and parallel in/serial out functions.

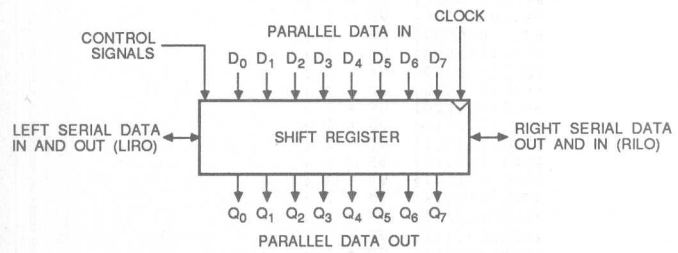


Figure 41. A Shift Register Block Diagram

Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0	I1	I0	
Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0	0	0	;Retain Data
RILO	Q7	Q6	Q5	Q4	Q3	Q2	Q1	0	1	;Shift Right
Q6	Q5	Q4	Q3	Q2	Q1	Q0	LIRO	1	0	;Shift Left
D7	D6	D5	D4	D3	D2	D1	D0	1	1	;Load Data

Figure 42. The Truth Table for a Universal Shift Register

```
TITLE           UNIVERSAL_SHIFT_REGISTER
PATTERN         SHIFT.PDS
REVISION        01
AUTHOR          JOE ENGINEER
COMPANY         MONOLITHIC MEMORIES, INC.
DATE            09/12/87

CHIP UNIV_SHI  PAL20X8

CLK   I0   D0 D1 D2 D3 D4 D5 D6 D7   I1   GND
/OC RILO Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0 LIRO VCC

;DESCRIPTION

;THIS PAL DEVICE IS AN 8-BIT SHIFT REGISTER WITH PARALLEL LOAD AND
;HOLD CAPABILITY.  TWO FUNCTION SELECT INPUTS (I0,I1) PROVIDE ONE OF
;FOUR OPERATIONS WHICH OCCUR SYNCHRONOUSLY ON THE RISING EDGE OF THE
;CLOCK (CLK).  THESE OPERATIONS ARE:

;       /OC   CLK   I1   I0   D7-D0   Q7-Q0      OPERATION
;       ------------------------------------------------
;        H     X    X    X      X       Z        HI-Z
;        L     C    L    L      X       L        HOLD
;        L     C    L    H      X     SR(Q)      SHIFT RIGHT
;        L     C    H    L      X     SL(Q)      SHIFT LEFT
;        L     C    H    H      D       D        LOAD
;       ------------------------------------------------

;TWO OR MORE OCTAL SHIFT REGISTERS MAY BE CASCADED TO PROVIDE LARGER
;SHIFT REGISTERS.  RILO AND LIRO ARE LOCATED ON PINS 14 AND 23
;RESPECTIVELY, WHICH PROVIDES FOR CONVENIENT INTERCONNECTIONS WHEN
;TWO OR MORE OCTAL SHIFT REGISTERS ARE CASCADED TO IMPLEMENT LARGER
;SHIFT REGISTERS.

EQUATIONS

/Q0 :=  /I1*/I0*/Q0                    ;HOLD Q0
     +  /I1* I0*/Q1                    ;SHIFT RIGHT
    :+:  I1*/I0*/LIRO                  ;SHIFT LEFT
     +   I1* I0*/D0                    ;LOAD D0

/Q1 :=  /I1*/I0*/Q1                    ;HOLD Q1
     +  /I1* I0*/Q2                    ;SHIFT RIGHT
    :+:  I1*/I0*/Q0                    ;SHIFT LEFT
     +   I1* I0*/D1                    ;LOAD D1

/Q2 :=  /I1*/I0*/Q2                    ;HOLD Q2
     +  /I1* I0*/Q3                    ;SHIFT RIGHT
    :+:  I1*/I0*/Q1                    ;SHIFT LEFT
     +   I1* I0*/D2                    ;LOAD D2

/Q3 :=  /I1*/I0*/Q3                    ;HOLD Q3
     +  /I1* I0*/Q4                    ;SHIFT RIGHT
    :+:  I1*/I0*/Q2                    ;SHIFT LEFT
     +   I1* I0*/D3                    ;LOAD D3

/Q4 :=  /I1*/I0*/Q4                    ;HOLD Q4
     +  /I1* I0*/Q5                    ;SHIFT RIGHT
    :+:  I1*/I0*/Q3                    ;SHIFT LEFT
     +   I1* I0*/D4                    ;LOAD D4

/Q5 :=  /I1*/I0*/Q5                    ;HOLD Q5
     +  /I1* I0*/Q6                    ;SHIFT RIGHT
    :+:  I1*/I0*/Q4                    ;SHIFT LEFT
     +   I1* I0*/D5                    ;LOAD D5
```

Figure 43. Octal Shift Register Design File

```
/Q6 :=  /I1*/I0*/Q6                    ;HOLD Q6
     +  /I1* I0*/Q7                    ;SHIFT RIGHT
    :+:  I1*/I0*/Q5                    ;SHIFT LEFT
     +   I1* I0*/D6                    ;LOAD D6

/Q7 :=  /I1*/I0*/Q7                    ;HOLD Q7
     +  /I1* I0*/RILO                  ;SHIFT RIGHT
    :+:  I1*/I0*/Q6                    ;SHIFT LEFT
     +   I1* I0*/D7                    ;LOAD D7

        /LIRO = /Q0                    ;LEFT IN RIGHT OUT
   LIRO.TRST = /I1*I0

        /RILO = /Q7                    ;RIGHT IN LEFT OUT
   RILO.TRST = I1*/I0

; SIMULATION NOT SHOWN HERE
```

Figure 43. Octal Shift Register Design File (Cont'd.)

Barrel Shifters

In most data processing systems, some form of data shifting or rotation is necessary. In typical computer systems, the shifter is located at the output of the ALU, and usually requires a single-cycle shift and add function (Figure 44). For such applications as floating-point arithmetic or string manipulation, ordinary shift registers are inefficient, since they require n clock cycles for an n-bit shift.

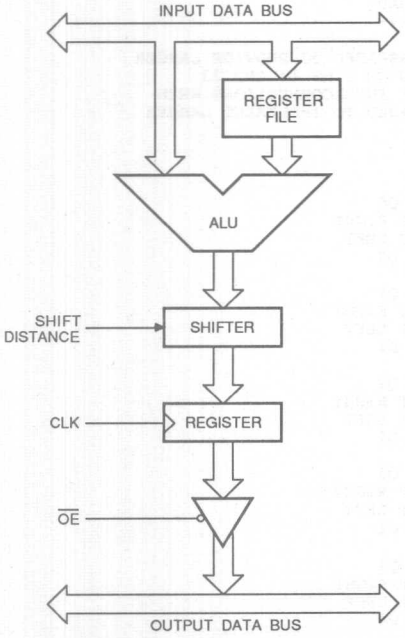


Figure 44. Typical ALU Architecture

A specialized shift register, called a "barrel shifter," is used to shift (or rotate) data by any number of bits in a single clock cycle. The name "barrel shifter" is used because of the circular nature of the shift operation. The storage registers on the output of the shifter are used in this architecture to pipeline the data operation, increasing throughput. The three-state buffer on the output registers is also useful for providing an interface to the data bus.

The design of a barrel shifter proceeds in the same manner as a regular shift register. The truth table is drawn, and the Boolean equations are then written based upon the truth tables. An eight-bit barrel shifter requires at least eight data inputs, eight registered data outputs, three control lines to specify the shift distance, a clock input and an output enable that controls the three-state buffer on the register output.

Figure 45 shows the block diagram for an eight-bit registered barrel shifter, while Figure 46 shows the truth table. The registered barrel shifter requires a total of 14 inputs and 8 outputs. This function can be easily implemented in an AmPAL22V10. Figure 47 shows the PLPL design file for this eight-bit barrel shifter function.

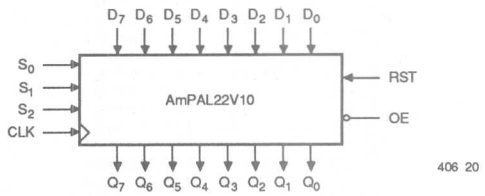


Figure 45. Block Diagram of an Eight-bit Barrel Shifter

S2	S1	S0	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
0	0	0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	D6	D5	D4	D3	D2	D1	D0	D7
0	1	0	D5	D4	D3	D2	D1	D0	D7	D6
0	1	1	D4	D3	D2	D1	D0	D7	D6	D5
1	0	0	D3	D2	D1	D0	D7	D6	D5	D4
1	0	1	D2	D1	D0	D7	D6	D5	D4	D3
1	1	0	D1	D0	D7	D6	D5	D4	D3	D2
1	1	1	D0	D7	D6	D5	D4	D3	D2	D1

Figure 46. Truth Table for an Eight-bit Barrel Shifter

```
DEVICE   EIGHT_BIT_BARREL_SHIFTER  (PAL22V10)

PIN      CLK      = 1
         D [7:0] = 2:9  /Q[7:0] = 23:16
         SEL2     = 13    SEL1    = 11     SEL0 = 10
         OE       = 14    RSY     = 15;

BEGIN
    IF (RST) THEN ARESET();       "SHIFTER INITIALIZATION"
    IF (OE)  THEN ENABLE(Q[7:0]); "ENABLE OUTPUTS"
    IF (/SEL2 * /SEL1 * /SEL0) THEN Q[7:0] := D[7:0];
    IF (/SEL2 * /SEL1 *  SEL0) THEN Q[7:0] := D[6:0], D[7];
    IF (/SEL2 *  SEL1 * /SEL0) THEN Q[7:0] := D[5:0], D[7:6];
    IF (/SEL2 *  SEL1 *  SEL0) THEN Q[7:0] := D[4:0], D[7:5];
    IF ( SEL2 * /SEL1 * /SEL0) THEN Q[7:0] := D[3:0], D[7:4];
    IF ( SEL2 * /SEL1 *  SEL0) THEN Q[7:0] := D[2:0], D[7:3];
    IF ( SEL2 *  SEL1 * /SEL0) THEN Q[7:0] := D[1:0], D[7:2];
    IF ( SEL2 *  SEL1 *  SEL0) THEN Q[7:0] := D[0], D[7:1];

END.

"FUNCTION TABLE SPECIFICATION"

TEST_VECTORS

IN CLK RST D[7:0] SEL2 SEL1 SEL0;
I_O OE;
OUT Q[7:0];
BEGIN
"CLK RST D[7:0]     SEL2  SEL1  SEL0  OE  Q[7:0]  "
"_____________________________________________"

C    1   00000000    1     0     1     0   ZZZZZZZZ;
C    0   11111111    0     1     0     1   HHHHHHHH;
C    1   XXXXXXXX    X     X     X     1   LLLLLLLL;
C    0   XXXXXXXX    X     X     X     0   ZZZZZZZZ;
C    0   00000000    0     0     0     1   LLLLLLLL;
C    0   11111111    0     0     0     1   HHHHHHHH;
C    0   01111111    1     0     0     1   HHHHLHHH;
C    0   01111111    0     1     0     1   HHHHHHLH;
C    0   01111111    1     1     0     1   HHLHHHHH;

END.
```

Figure 47. Design File for an Eight-bit Barrel Shifter

Gray-code, Johnson Counter and Shift Register Device Selection Considerations

Gray-code counters, Johnson counters and shift registers are not very logic-intensive; the number of product terms required is minimal. The D-type flip-flops provide the most efficient implementations, allowing these designs to be easily implemented in most PAL devices.

Since Gray-code counters are often used as system clocks, very high speed PAL devices provide the highest resolution clocks.

Barrel shifters are very logic-intensive and require many product terms, since data from all the inputs needs to be accessible at any output. Registered PLDs with a large number of product terms are ideal for barrel shifters. Large barrel shifters can also be partitioned into a number of PLDs.

Until now we have discussed strictly synchronous registered designs, where a common system clock is used. In asynchronous registered designs, a common clock is not used. The register clock may be generated by the output of another register, or by a logical combination of various other signals. Such designs are usually slow for such applications as timing generation, because when the output of one register is used to clock another, multiple delays are encountered before all the register outputs stabilize. On the other hand, designs can be very fast for asynchronous applications such as bus arbitration and control, where a fast response to a bus signal can be provided without waiting for a common system clock.

Although asynchronous designs are easier to visualize, they present larger problems in implementation. Combinatorial hazworry about race conditions when clocking a number of registers simultaneously. Careful design analysis is strongly recommended before implementing any asynchronous design.

Ripple counters are probably the easiest examples of such asynchronous designs. Figure 48 shows the logic diagram of a five-bit binary ripple counter. These counters clearly have the advantage of design simplicity. The output from one stage is fed as the clock to the next stage. However, this results in a slower counting rate, since the clock signals need to propagate through all five registers before the next count is reached.

The ripple counter design illustrated below is implemented in a PAL20RA10. Figure 49 shows the design file for this counter.

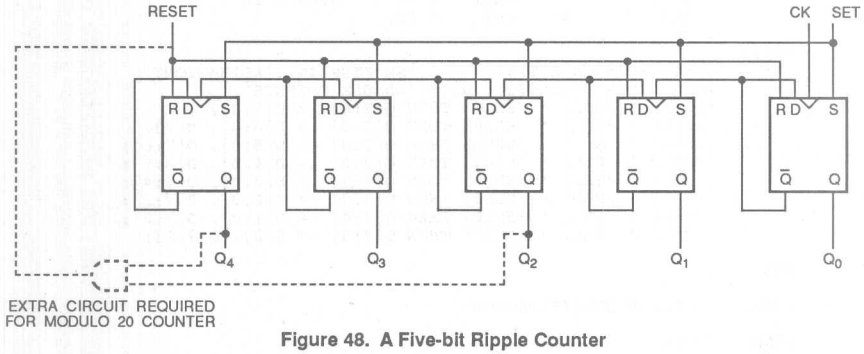


Figure 48. A Five-bit Ripple Counter

406 21

```
TITLE     Dual 5-Bit Counter
PATTERN   Upcount.pds
REVISION  01
AUTHOR    Joe Engineer
COMPANY   Monolithic Memories, Inc.
DATE      11/12/87

CHIP Upcount PAL20RA10

PL CKA CKB ENA ENB NC  NC  NC  NC  NC  INIT GND
OE QB4 QB3 QB2 QB1 QB0 QA4 QA3 QA2 QA1 QA0  VCC

EQUATIONS
; The First  Counter

/QA4          := QA4 * /ENA  ; Toggle if lower MSB
is zero
                + /QA4 * ENA   ; Hold count
QA4.CLKF      = /QA3         ; Toggle
QA4.RSTF      = INIT         ; Initialize
QA4.TRST      = VCC

/QA3          := QA3 * /ENA  ; Toggle when QA2 zero
                + /QA3 * ENA   ; Hold count
QA3.CLKF      = /QA2         ; Toggle
QA3.RSTF      = INIT         ; Initialize
QA3.TRST      = VCC
```

Figure 49. Design File for a Five-bit Ripple Counter

```
/QA2          := QA2 * /ENA  ; Toggle when QA1 zero
               + /QA2 * ENA  ; Hold count
QA2.CLKF      = /QA1         ; Toggle
QA2.RSTF      = INIT         ; Initialize
QA2.TRST      = VCC

/QA1          := QA1 * /ENA  ; Toggle when QA0 zero
               + /QA1 * ENA  ; Hold count
QA1.CLKF      = /QA0         ; Toggle
QA1.RSTF      = INIT         ; Initialize
QA1.TRST      = VCC

/QA0          := QA0 * /ENA  ; Toggle LSB
               + /QA0 * ENA  ; Hold count
QA0.CLKF      = CKA          ; Toggle
QA0.RSTF      = INIT         ; Initialize
QA0.TRST      = VCC


; The Second Counter

/QB4          := QB4 * /ENB  ; Toggle if lower MSB is zero
               + /QB4 * ENB  ; Hold count
QB4.CLKF      = /QB3         ; Toggle
QB4.RSTF      = INIT         ; Initialize
QB4.TRST      = VCC

/QB3          := QB3 * /ENB  ; Toggle when QB2 zero
               + /QB3 * ENB  ; Hold count
QB3.CLKF      = /QB2         ; Toggle
QB3.RSTF      = INIT         ; Initialize
QB3.TRST      = VCC

/QB2          := QB2 * /ENB  ; Toggle when QB1 zero
               + /QB2 * ENB  ; Hold count
QB2.CLKF      = /QB1         ; Toggle
QB2.RSTF      = INIT         ; Initialize
QB2.TRST      = VCC

/QB1          := QB1 * /ENB  ; Toggle when QB0 zero
               + /QB1 * ENB  ; Hold count
QB1.CLKF      = /QB0         ; Toggle
QB1.RSTF      = INIT         ; Initialize
QB1.TRST      = VCC

/QB0          := QB0 * /ENB  ; Toggle LSB
               + /QB0 * ENB  ; Hold count
QB0.CLKF      = CKB          ; Toggle
QB0.RSTF      = INIT         ; Initialize
QB0.TRST      = VCC

SIMULATION

TRACE_ON /PL QA4 QA3 QA2 QA1 QA0 QB4 QB3 QB2 QB1 QB0

SETF PL INIT /CKA /CKB /OE
SETF INIT CKA CKB
SETF /INIT /CKA /CKB
CHECK /QA4 /QA3 /QA2 /QA1 /QA0
CHECK /QB4 /QB3 /QB2 /QB1 /QB0
```

Figure 49. Design File for a Five-bit Ripple Counter (Cont'd.)

```
SETF CKA CKB
CHECK /QA4 /QA3 /QA2 /QA1  QA0
CHECK /QB4 /QB3 /QB2 /QB1  QB0
SETF /CKA /CKB

SETF CKA CKB
CHECK /QA4 /QA3 /QA2  QA1 /QA0
CHECK /QB4 /QB3 /QB2  QB1 /QB0
SETF /CKA /CKB

SETF CKA CKB
CHECK /QA4 /QA3 /QA2  QA1  QA0
CHECK /QB4 /QB3 /QB2  QB1  QB0
SETF /CKA /CKB

SETF CKA CKB
CHECK /QA4 /QA3  QA2 /QA1 /QA0
CHECK /QB4 /QB3  QB2 /QB1 /QB0
SETF /CKA /CKB

SETF CKA CKB
CHECK /QA4 /QA3  QA2 /QA1  QA0
CHECK /QB4 /QB3  QB2 /QB1  QB0
SETF /CKA /CKB

TRACE_OFF
```

Figure 49. Design File for a Five-bit Ripple Counter (Cont'd.)

There are additional independent SET and RESET controls in the PAL20RA10 which can be used to restart the count. If a modulo counter is desired, these SET and RESET signals can be used as described above. Figure 48 shows the implementation of a modulo-20 counter which is RESET when output bits Q4 and Q2 are both HIGH. Since the RESET is implemented with a product term, the extra AND gate shown can be implemented directly within the PAL device.

Asynchronous Designs Device Selection Considerations

The device selection for asynchronous designs is easy. As the clock signals require logic, only PLDs which allow implementation of Boolean logic on the clock signals are useful. Figure 50 shows a list of devices which provide programmable clock signals. They also provide programmable SET and RESET product terms, and the capability of bypassing the register.

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16RA8 20RA10	20 24	8 10	0-8 0-10	8-0 10-0	4 4	20 33, 20
CMOS						
C29MA16	24	5	-	16	4-12	20, 15

Figure 50. Programmable Logic Devices for Asynchronous Designs

Other Applications of Registered PLDs

Registered PLDs are used for a number of miscellaneous applications which are not covered by the synchronous and asynchronous design applications discussed up to now. In the subsequent discussion we will cover a few of these applications:

- Frequency dividers
- Addressable registers

Frequency Dividers

Standard synchronous counters provide the basic capability of dividing an input frequency. A single register of a PAL device will let us divide by two. If we stack these registers, a binary counter provides symmetrical division by 2,4,8,16, etc. This divider has been a standard for years, and the PAL device has always been an excellent choice for such applications.

One unique application of PAL devices is for dividing input frequencies by odd numbers. This has been done historically by designing a counter which cycles modulo an odd number, and decoding the specific states of the counter. The disadvantage of this approach is that the output is not symmetrical and the duty cycle is not 50%.

Let's examine a simple divide-by-five counter. This counter can be implemented using three flip-flops that start at zero and reset at four, resulting in a five-state counter. The table in Figure 51 shows the outputs of the three individual flip-flops.

Present State			Next State			
Q2	Q1	Q0	Q2	Q1	Q0	
0	0	0	0	0	1	State zero to one.
0	0	1	0	1	0	State one to two.
0	1	0	0	1	1	State two to three.
0	1	1	1	0	0	State three to four.
1	0	0	0	0	0	State four to zero.

Figure 51. Truth Table for a Five-bit Counter

The Boolean equations are:

```
Q2 := /Q2 *  Q1 * Q0   ;MSB bit
Q1 := /Q1 *  Q0 + Q1 * /Q0
Q0 := /Q2 * /Q0        ;LSB bit
```

The waveforms for this divider are shown in Figure 52. Notice that the Q2 output goes HIGH for one state and that this output is one-fifth of the input frequency, but it is a 20% duty cycle. Q1 is active for two states; it provides the same frequency, but with a 40% duty cycle. If we want a 50% duty cycle we are going to have to divide a state in half.

To provide the 50% duty cycle, the two edges should be evenly spaced in the count sequence, one edge in the middle of state two and one at the beginning of state zero. The first edge can be formed by logically "ANDing" state_2 with the falling edge of the clock. The second edge can be formed by decoding state zero.

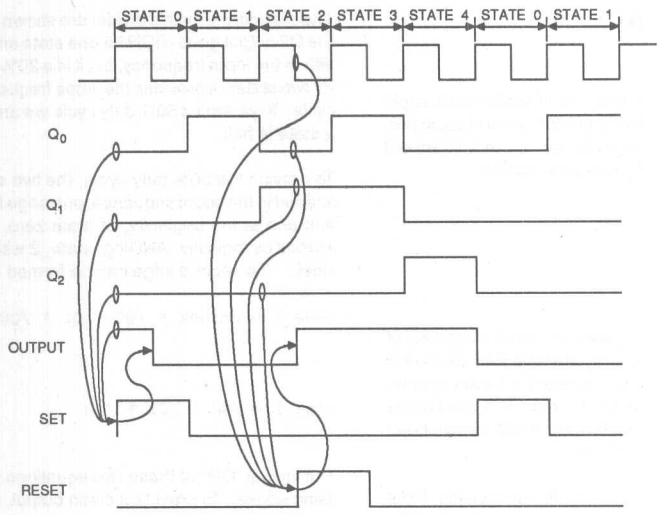
```
edge_1 = /clock * /Q2 * Q1 * /Q0   ;edge between
                                   ;states two and
                                   ;three

edge_2 = /Q2 * /Q1 * /Q0           ;edge at state
                                   ;zero
```

The logical "OR" of these two equations will provide the needed rising edges. To provide a clean output, this signal should clock another output register.

The next step in the design is to pick the appropriate PAL device to fit this design. Our biggest concern is that we need the capability of clocking the counter at one speed and the output flip-flop at another. To do this, we cannot use a PAL device that has a dedicated clock pin; we need an architecture that allows programmable clocks. The PAL20RA10, PAL16RA8 and AmPALC29MA16 have this feature. This design uses the PAL20RA10 device.

Since the clock signal requires two product terms (one for each edge), and the PAL20RA10 clock signal has only one product term, this implementation is not possible. Another technique is to use the independent asynchronous SET and asynchronous RESET product terms of the PAL20RA10 output register. A HIGH on the SET product term asserts the register output, and a HIGH on the RESET product term unasserts the register output. Due to the asynchronous nature of the product terms some adjustment in timing is required. The SET product term is asserted when in state 0 (Q2=0, Q1=0 and Q0=0), and the RESET product term is asserted when between states two and three.

```
OUTPUT.SET = /clock * /Q2 * Q1 * /Q0 ;set between
                                     ;states 2 & 3
OUTPUT.RESET = /Q2 * /Q1 * /Q0       ;reset at
                                     ;state zero
```



406 22

Figure 52. Waveform for a Frequency Divider

```
TITLE           DIVIDE BY FIVE EXAMPLE
PATTERN         DIVBY5.PDS
REVISION        A
AUTHOR          BARRY SEIDNER
COMPANY         MONOLITHIC MEMORIES INC., SANTA CLARA, CA
DATE            7/25/1987

CHIP DIVIDER PAL20RA10

PL CLK I3  I4  I5  I6  I7  I8   I9   I10  I11  GND
OE Q14 Q15 Q16 Q17 Q18 Q19 Q20  /Q21 /Q22 /Q23 VCC

EQUATIONS

;The following equations delineate the three bit counter of our
;state machine driver. It has a total count of five starting
;at zero and counting thru four, then resetting to zero.

Q23:= Q21 * Q22 * /Q23                  ;MSB OF STATE MACHINE
Q23.CLKF=CLK                            ;CLOCK ON RISING EDGE OF CLK PIN
Q23.TRST=VCC                            ;OUTPUT ALWAYS ON

Q22:=/Q22 * Q21 + Q22 * /Q21            ;BIT 1 OF STATE MACHINE
Q22.CLKF=CLK                            ;CLOCK ON RISING EDGE OF CLK PIN
Q22.TRST=VCC                            ;OUTPUT ALWAYS ON

Q21:=/Q21 * /Q23                        ;LSB OF STATE MACHINE
Q21.CLKF=CLK                            ;CLOCK ON RISING EDGE OF CLK
Q21.TRST=VCC                            ;OUTPUT ALWAYS ON


;The last cell of our divide by five counter is the actual
;output of divider. It divides by setting and resetting at the
;appropriate time in the count sequence. The output will stay high
;for count zero, one, and the first half of two-then go low for
;the second half of two, thru three and four.

Q20.SETF = /Q23 *  Q22 * /Q21 * /CLK    ;SET ON COUT THREE AND 1/2
Q20.RSTF = /Q23 * /Q22 * /Q21           ;RESET ON COUNT ZERO
Q20.CLKF = GND                          ;CLOCK INPUT NOT USED
Q20      := GND                         ;DATA INPUT NOT USED
Q20.TRST = VCC                          ;OUTPUT ALWAYS ON


SIMULATION

;This simulation will run the divider through
;several passes of the count to verify operation.

TRACE_ON    OE PL CLK Q23 Q22 Q21 Q20

;Initialization of all inputs. Before the simulator can be
;executed, all of the inputs to the PAL device should be
;set to a specific value. This section of simulator
;declaration sets three input variables inactive.

SETF    PL /OE  /CLK                    ;SET PRELOAD FALSE
                                        ;SET OUTPUT ENABLE
                                        ;SET CLOCK LOW

;The PRLDF statement will set the registers to a known
;state at the beginning of our simulation. This line sets
;us to state zero.

PRLDF   /Q23 /Q22 /Q21 Q20

;The FOR loop statement exemplifies an easy way to perform
;repetitive functions. In this example, the "RA" cell does
;not have a dedicated clock input so this routine will
;generate 30 repetitive clocks to drive our design.

FOR I:=1 TO 30 DO
   BEGIN
      SETF  CLK                         ;SET CLOCK HIGH
      SETF /CLK                         ;SET CLOCK LOW
   END

;End of simulation
TRACE_OFF
```

Figure 53. Design File for Frequency Divider

Addressable registers are commonly-used MSI functions, often implemented in PAL devices. Addressable registers are used as building blocks for digital computers. Depending upon the address, they can load data into one of several registers or they can output the contents of one of several registers. Addressable registers can also be used to store previous values. An example of a 16-bit addressable register is shown in Figure 54.

```
TITLE     16-BIT ADDRESSABLE REGISTER
PATTERN   ADREG16.PDS
REVISION  A
AUTHOR    John Birkner
COMPANY   Monolithic Memories Inc. Santa Clara, CA
DATE      2/11/85

;         The 16-bit addressable register loads one of 16 registers
;         selected by ADDR[0..3] with data input, DATA.


CHIP ADREG16 PAL32R16
Q0 Q1 Q2 Q3 /E1 NC NC A0 A1 VCC A2 A3 DATA NC /PRLD2 CLK2
Q4 Q5 Q6 Q7 Q8 Q9 Q10 Q11 /E2 NC NC NC NC GND NC NC NC NC /PRLD1 CLK1
Q12 Q13 Q14 Q15


EQUATIONS

Q0      :=  A0              *Q0           ;hold
        +       A1          *Q0           ;hold
        +           A2      *Q0           ;hold
        +               A3*Q0             ;hold
        +  /A0*/A1*/A2*/A3*DATA           ;load

Q1      := /A0              *Q1           ;hold
        +       A1          *Q1           ;hold
        +           A2      *Q1           ;hold
        +               A3*Q1             ;hold
        +   A0*/A1*/A2*/A3*DATA           ;load

Q2      :=  A0              *Q2           ;hold
        +      /A1          *Q2           ;hold
        +           A2      *Q2           ;hold
        +               A3*Q2             ;hold
        +  /A0* A1*/A2*/A3*DATA           ;load

Q3      := /A0              *Q3           ;hold
        +       A1          *Q3           ;hold
        +           A2      *Q3           ;hold
        +               A3*Q3             ;hold
        +   A0* A1*/A2*/A3*DATA           ;load

Q4      :=  A0              *Q4           ;hold
        +       A1          *Q4           ;hold
        +          /A2      *Q4           ;hold
        +               A3*Q4             ;hold
        +  /A0*/A1* A2*/A3*DATA           ;load

Q5      := /A0              *Q5           ;hold
        +       A1          *Q5           ;hold
        +          /A2      *Q5           ;hold
        +               A3*Q5             ;hold
        +   A0*/A1* A2*/A3*DATA           ;load

Q6      :=  A0              *Q6           ;hold
        +      /A1          *Q6           ;hold
        +          /A2      *Q6           ;hold
        +               A3*Q6             ;hold
        +  /A0* A1* A2*/A3*DATA           ;load

Q7      := /A0              *Q7           ;hold
        +      /A1          *Q7           ;hold
        +          /A2      *Q7           ;hold
        +               A3*Q7             ;hold
        +   A0* A1* A2*/A3*DATA           ;load

Q8      :=  A0              *Q8           ;hold
        +       A1          *Q8           ;hold
        +           A2      *Q8           ;hold
        +              /A3*Q8             ;hold
        +  /A0*/A1*/A2* A3*DATA           ;load

Q9      := /A0              *Q9           ;hold
        +       A1          *Q9           ;hold
        +           A2      *Q9           ;hold
        +              /A3*Q9             ;hold
        +   A0*/A1*/A2* A3*DATA           ;load

Q10     :=  A0              *Q10          ;hold
        +      /A1          *Q10          ;hold
        +          A2       *Q10          ;hold
        +              /A3*Q10            ;hold
        +  /A0* A1*/A2* A3*DATA           ;load

Q11     := /A0              *Q11          ;hold
        +      /A1          *Q11          ;hold
        +          A2       *Q11          ;hold
        +              /A3*Q11            ;hold
        +   A0* A1*/A2* A3*DATA           ;load

Q12     :=  A0              *Q12          ;hold
        +       A1          *Q12          ;hold
        +          /A2      *Q12          ;hold
        +              /A3*Q12            ;hold
        +  /A0*/A1* A2* A3*DATA           ;load

Q13     := /A0              *Q13          ;hold
        +       A1          *Q13          ;hold
        +          /A2      *Q13          ;hold
        +              /A3*Q13            ;hold
        +   A0*/A1* A2* A3*DATA           ;load

Q14     :=  A0              *Q14          ;hold
        +      /A1          *Q14          ;hold
        +          /A2      *Q14          ;hold
        +              /A3*Q14            ;hold
        +  /A0* A1* A2* A3*DATA           ;load

Q15     := /A0              *Q15          ;hold
        +      /A1          *Q15          ;hold
        +          /A2      *Q15          ;hold
        +              /A3*Q15            ;hold
        +   A0* A1* A2* A3*DATA           ;load

SIMULATION

TRACE_ON Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15
         A0 A1 A2 A3 DATA

SETF E1 E2 /DATA /PRLD1 /PRLD2

       SETF /A0 /A1 /A2 /A3
       CLOCKF CLK1 CLK2

       SETF  A0 /A1 /A2 /A3
       CLOCKF CLK1 CLK2

       SETF /A0  A1 /A2 /A3
       CLOCKF CLK1 CLK2

       SETF  A0  A1 /A2 /A3
       CLOCKF CLK1 CLK2

       SETF /A0 /A1  A2 /A3
       CLOCKF CLK1 CLK2

       SETF  A0 /A1  A2 /A3
       CLOCKF CLK1 CLK2

       SETF /A0  A1  A2 /A3
       CLOCKF CLK1 CLK2

       SETF  A0  A1  A2 /A3
       CLOCKF CLK1 CLK2

       SETF /A0 /A1 /A2  A3
       CLOCKF CLK1 CLK2

       SETF  A0 /A1 /A2  A3
       CLOCKF CLK1 CLK2

       SETF /A0  A1 /A2  A3
       CLOCKF CLK1 CLK2

       SETF  A0  A1 /A2  A3
       CLOCKF CLK1 CLK2

       SETF /A0 /A1  A2  A3
       CLOCKF CLK1 CLK2

       SETF  A0 /A1  A2  A3
       CLOCKF CLK1 CLK2

       SETF /A0  A1  A2  A3
       CLOCKF CLK1 CLK2

       SETF  A0  A1  A2  A3
       CLOCKF CLK1 CLK2

       SETF DATA

       SETF /A0 /A1 /A2 /A3
       CLOCKF CLK1 CLK2


Page :  1
        g g cgcgcg cgcgcgcgcg cgcgcgcgcg cgcgcgc
  Q0    XXXXLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLH
  Q1    XXXXXLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLL
  Q2    XXXXXXXXLL LLLLLLLLLL LLLLLLLLLL LLLLLL
  Q3    XXXXXXXXXX LLLLLLLLLL LLLLLLLLLL LLLLLL
  Q4    XXXXXXXXXX XXLLLLLLLL LLLLLLLLLL LLLLLL
  Q5    XXXXXXXXXX XXXXLLLLLL LLLLLLLLLL LLLLLL
  Q6    XXXXXXXXXX XXXXXXLLLL LLLLLLLLLL LLLLLL
  Q7    XXXXXXXXXX XXXXXXXXLL LLLLLLLLLL LLLLLL
  Q8    XXXXXXXXXX XXXXXXXXXX LLLLLLLLLL LLLLLL
  Q9    XXXXXXXXXX XXXXXXXXXX XXLLLLLLLL LLLLLL
  Q10   XXXXXXXXXX XXXXXXXXXX XXXXLLLLLL LLLLLL
  Q11   XXXXXXXXXX XXXXXXXXXX XXXXXXLLLL LLLLLL
  Q12   XXXXXXXXXX XXXXXXXXXX XXXXXXXXLL LLLLLL
  Q13   XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX LLLLLL
  Q14   XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX XXLLLL
  Q15   XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX XXXXLL
  A0    XXLLLHHLLH HLHHLLHHL LHHLLHHLLH HLLHHLL
  A1    XXLLLLLHHH HLLLLHHHHL LLLHHHHLLL LHHHHLL
  A2    XXLLLLLLLL LHHHHHHHHL LLLLLLHHHH HHHHHLL
  A3    XXLLLLLLLL LLLLLLLLLH HHHHHHHHHH HHHHHLL
  DATA  LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLHH
```

Figure 54. A Sixteen-bit Addressable Register Design File

State Machine Design

Introduction

State machine designs are widely used for sequential control logic, which forms the core of many digital systems. State machines are required in a variety of applications covering a broad range of performance and complexity; low-level control of microprocessor-to-VLSI-peripheral interfaces, bus arbitration and timing generation in conventional microprocessors, custom bit-slice microprocessors, data encryption and decryption and transmission protocols are but a few examples.

Typically, the details of control logic are the last to be settled in the design cycle, since they are continuously affected by changing system requirements and feature enhancements. Programmable logic is a forgiving solution for control logic design because it allows easy modifications to be made without disturbing PC board layout. Its flexibility provides an escape valve that permits design changes without impacting time-to-market.

A majority of registered PAL device applications are sequential control designs where state machine design techniques are employed. As technology advances, new high-speed and high-functionality devices are being introduced which simplify the task of state machine design. A broad spectrum of different functionality-and-performance solutions is available for state machine design. In this discussion we will examine the functions performed by state machines, their implementation on various devices, and their selection. Finally, we will implement a state machine design and go through all of the stages involved in a design tutorial.

What is a State Machine?

A state machine is a digital device which traverses through a predetermined sequence of states in an orderly fashion. A state is a set of values measured at different parts of the circuit. A simple state machine can consist of PAL-device-based combinatorial logic, output registers, and buried (state) registers. The state in such a sequencer is determined by the values stored in the buried and/or output registers.

A general form of a state machine can be depicted as a device shown in Figure 1. In addition to the device inputs and outputs, a state machine consists of two essential elements: combinatorial logic and memory (registers). This is similar to the registered counter designs discussed on page 2-66, which are essentially simple state machines. The memory is used to store the state of the machine. The combinatorial logic can be viewed as two distinct functional blocks: the next state decoder and the output decoder (Figure 2). The next state decoder determines the next state of the state machine while the output decoder generates the actual outputs. Although they perform two distinct functions, these are usually combined into one combinatorial logic array as in Figure 1.

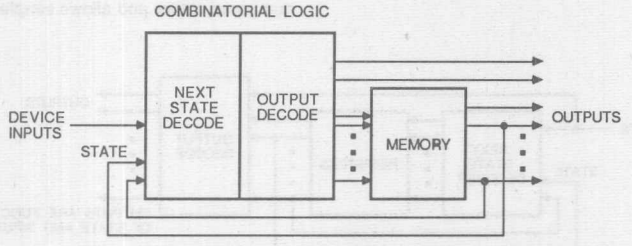


Figure 1. Block Diagram of a Simple State Machine

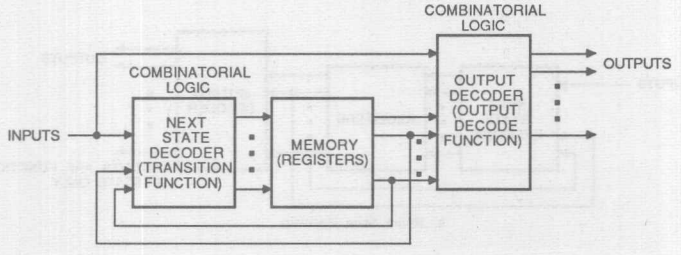


Figure 2. State Machine, with Separate Output & Next State Decoders

The basic operation of a state machine is twofold:

1. It traverses through a sequence of states, where the next state is determined by next state decoder, depending upon the present state and input conditions.

2. It provides sequences of output signals based upon state transitions. The outputs are generated by the output decoder based upon present state and input conditions.

Using input signals for deciding the next state is also known as branching. In addition to branching, complex sequencers provide the capability of repeating sequences (looping) and subroutines. The transitions from one state to another are called *control sequencing* and the logic required for deciding the next states is called the *transition function* (Figure 2).

The use of input signals in the decision-making process for *output generation* determines the type of a state machine. There are two widely-known types of state machines: Mealy and Moore (Figure 3). Moore state machine outputs are a function of the present state only. In the more general Mealy-type state machine, the outputs are functions of both the state and the input signals. The logic required is known as the *output function*. For either type, the control sequencing depends upon both states and input signals.

Most practical state machines are synchronous sequential circuits which rely on clock signals to trigger the state transitions. A single clock is connected to all of the state and output edge-triggered flip-flops, which allows a state change to occur on the rising edge of the clock. Asynchronous state machines are also possible, which utilize the propagation delay in combinatorial logic for the memory function of the state machine. Such machines are highly susceptible to hazards, hard to design and are seldom used. In our discussion we will focus solely on sequential state machines.

State Machine Applications

State machines are used in a number of system control applications. A sampling of a few of the applications, and how state machines are applied, is described below.

As sequencers for digital signal processing (DSP) applications, state machines offer speed and sufficient functionality without the overkill of complex microprocessors. For simple algorithms, such as those involved in performing a Fast Fourier Transform (FFT), a state machine can control the set of vectors that are multiplied and added in the process. For complex DSP operations, a programmable DSP may be better. On the other hand, the programmable DSP solution is not likely to be as fast as the dedicated hardware approach.

Consider the case of a video controller. It generates addresses for scanning purposes, using counters with various sequences and lengths. But instead of implementing these as actual counters, the sequences involved can be "unlocked" and implemented, instead, as state machine transitions. And there is an advantage beyond mere economy of parts. A count can be set or initiated, then left to take care of itself, freeing the microprocessor for other operations.

In peripheral control the simple state machine approach can be very efficient. Consider the case of run-length-limited (RLL) code. Both encoding and decoding can be translated into state machines, which examine the serial data stream as it is read, and generate the output data.

Industrial control and robotics offer further areas where simple control functions are required. Such tasks as mechanical positioning of a robot arm, simple decision making, and calculation of a trigonometric function, usually do not require the high-power solution of microprocessors with stacks and pointers. Rather, what is required is a device that is capable of storing a limited number of states and allows simple branching upon conditions.

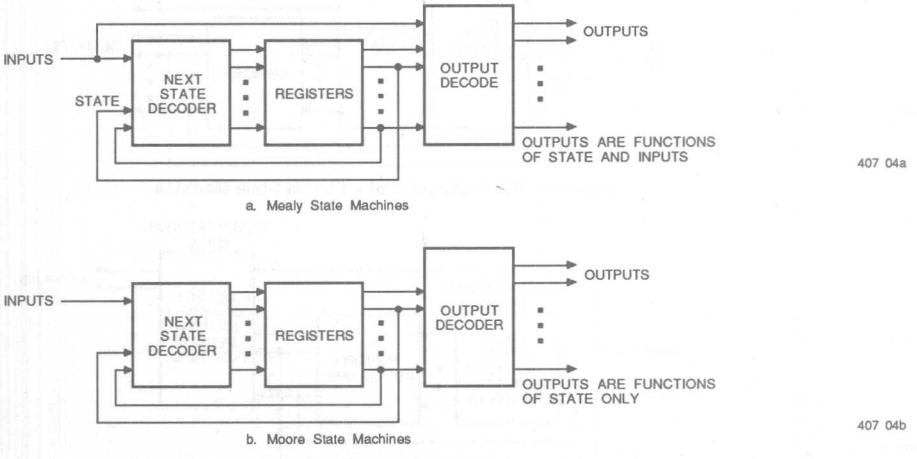


Figure 3. The Two Standard State Machine Models

Data encryption and decryption present similar problems to those encountered in encoding and decoding for mass media, only here it is desirable to make the scheme not so obvious. A programmable state machine device with a security fuse is ideal for this because memory is internally programmed and cannot be accessed by someone tampering with the system.

Functions Performed

All of the system design functions performed by controllers can be categorized as one of the following state machine functions:

- Arbitration
- Event monitoring
- Multiple condition testing
- Timing delays
- Control signal generation

Later we will take a design example and illustrate how these functions can be used when designing a state machine.

State Machine Theory

Let us take a brief look at the underlying theory for all sequential logic systems, the *finite state machine* (FSM), or simply state machine.

Those parts of digital systems whose outputs depend on their past inputs as well as their current ones can be modeled as finite state machines. The "history" of the machine is summed up in the value of its internal state. When a new input is presented to the FSM, an output is generated which depends on this input and the present state of the FSM, and the machine is caused to move into a new state, referred to as the next state. This new state also depends on both the input and present state. The structure of an FSM is shown pictorially in Figure 2. The internal state is stored in a block labelled "memory". As discussed earlier, two combinatorial functions are required: the transition function, which generates the value of the next state, and the output function, which generates the state machine output.

State Diagram Representation

The behavior of an FSM may be specified in graphical form as shown in Figure 4. This is called a state diagram, or state transition diagram. Each bubble represents a state, and each arrow represents a transition between states. Inputs which cause the transitions are shown next to each transition arrow.

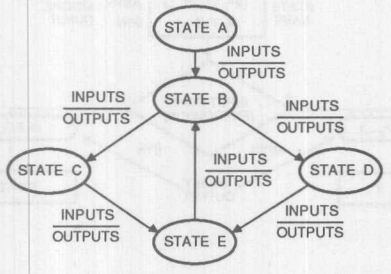


Figure 4. State Machine Representation

Control sequencing is represented in the state transition diagram as shown in Figure 5. Direct control sequencing requires an unconditional transition from state A to state B. Similarly conditional control sequencing shows a conditional transition from state C to either state D or state E, depending upon input signal I1.

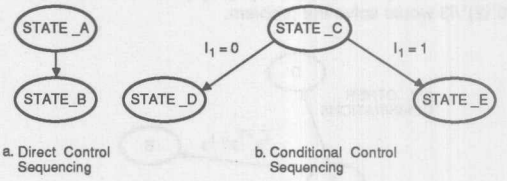


Figure 5. Control Sequencing

For Moore machines the output generation is represented by assigning outputs with states (bubbles) as shown in Figure 6. Similarly, for Mealy machines conditional output generation is represented by assigning outputs to transitions (arrows), as was shown in Figure 4. More detail on Mealy and Moore output generation is given later.

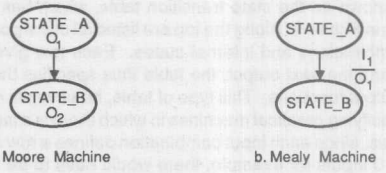


Figure 6. Output Generation

For this notation, there is a specification uncertainty as to which signals are outputs or inputs, as they both occur on the drawing next to the arrow in which they are active. This is usually resolved by separating the input and output signals names with a line (Figures 4 & 6). Sometimes an auxiliary pin list detailing the logic polarity and input or output designations is also used.

State transition diagrams can be made more compact by writing on the transitions not the input values which cause the transition, as in Figure 4, but a Boolean expression defining the input combination or combinations which cause this transition. For example, in Figure 7, some transitions have been shown for a machine with inputs "START", "X1" and "X2". In the transition between states 1 and 2, the inputs X1 and X2 are ignored (that is, they are "don't cares") and thus do not appear on the diagram. This saves space and makes the function more obvious.

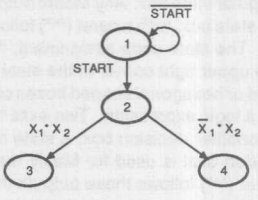


Figure 7. State Transition Diagram with Mnemonics

three input combinations, {I0, I1, I2, I3} = {1011}, {1101} and {1111}, which make both (/I0*/I2 + I3) and (I0*I1 + I0*I2) true. Since a transition to two next states is impossible, this is an error in the specification. It must either be guaranteed that these input combinations never occur, or the transition conditions must be modified. In this example, changing (I0*I1 + I0*I2) to (I0*I1 + I0*I2)*/I3 would solve the problem.

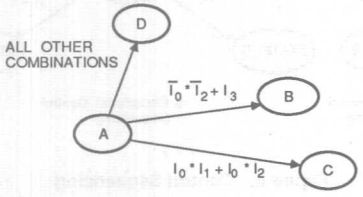


Figure 8. State Diagram with Conflicting Branch Conditions

State Transition Table Representation

A second method for state machine representation is the tabular form known as the state transition table, which has the format shown in Figure 9. Along the top are listed all of the possible input bit combinations and internal states. Each row gives the next state and the next output; the table thus specifies the transition and output functions. This type of table, however, is not suitable for specifying practical machines in which there is a large number of inputs, since each input combination defines a row of the table. With 10 inputs for example, there would have to be 1024 rows! A modified version of this table is often used directly for programmable logic sequencer (PLS) device design.

PRESENT STATE	INPUTS	NEXT STATE	OUTPUTS GENERATED
S_0-S_n	I_0-I_m	S_0-S_n	O_0-O_p

Figure 9. A State Transition Table

Flowcharts

Another popular notation is based on flowcharts. In this notation, states are represented by rectangular boxes, and alternative state transitions are determined by strings of diamond-shaped boxes. The elements may have multiple entry points, but in general have only one exit. The state name is written as the first entry in the rectangular state box. Any Moore outputs present are written next in the state box, with a caret ("^") following those that are unregistered. The state code assignment, if it is known, is written next to the upper right corner of the state box. Decision boxes are diamond or hexagonal shaped boxes containing either an input signal or a logic expression. Two exits labelled "0" and "1" lead to either another decision box, a state box, or a Mealy output. The rounded oval is used for Mealy machine outputs. Once again, a caret ("^") follows those outputs that are unregistered. All of the boxes may need to be expanded to accommodate a number of output signals or a larger expression.

particular combination or set of combinations of the input variables. A path does not have to include all input variables; thus it accommodates "don't cares". These decision trees take more space than the expressions would, but in many practical cases, state machine controllers only test a small subset of the input variables in each state and the trees are quite manageable. Also, the chain of decisions often mirrors the designer's way of thinking about the actions of the controller. It is important to note that these tests are not performed sequentially in the FSM; all are performed in parallel by the FSM's state transition logic.

A benefit of this method of specifying transitions is that the problem of Figure 8 can be avoided. Such a conflict would be impossible as one path cannot diverge to define paths to two states.

When there is no danger of conflicts due to multiple next states being defined, this flowchart notation can be compacted by allowing more complex decisions. Expressions can be tested, as shown in Figure 11a, or multiple branches can extend from a decoding box, as in Figure 11b. In the second case it is convenient to group the set of binary inputs into a vector, and branch on different values of this vector.

The three methods of state machine representation—state diagrams, state tables, and flowcharts—are all equivalent and interchangeable, since they all describe the same hardware structure. Each style has its own particular advantages. Although most popular, the state transition diagrams are more complex for problems where state transitions depend on many inputs, since the transition conditions are written directly on the transition arrows. Although cumbersome, the state tables allow the designer tight control over signal logic. Flowcharts are convenient for small problems where there are not more than about ten states and where up to two or three inputs or input expressions are tested in each state. For larger problems, they can become ungainly.

Once a state machine is defined, it must be implemented on a device. Software packages are then used to implement the design on a device. The task is to convert the state machine description into transition and output functions. Software packages also account for device-specific architectural variations and limitations, to provide a uniform user interface.

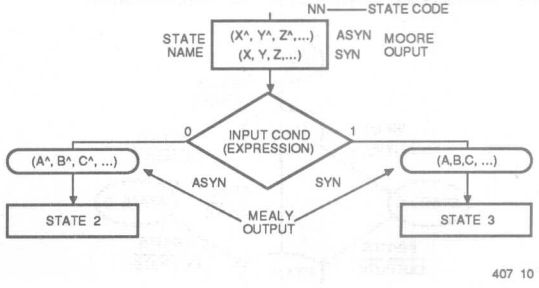


Figure 10. Flowchart Notation

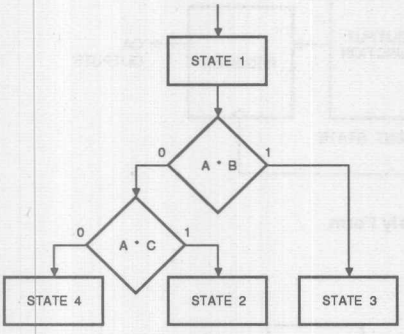
Some software packages accept all of the three different state machine representations directly as design inputs. However, the most prevalent design methodology is to convert the three state machine design representations to a simple textual representation. Textual representations are accepted by most software packages although the syntax varies. The PALASM 2 software package offers one such simple and easy-to-use state machine textual representation. The task of converting from a state transition diagram and flowchart representation to PALASM 2 software state machine syntax is demonstrated in a design tutorial on page 2-122.

Since the most common of all state machine representations is the state transition diagram representation, we will use it in all subsequent discussions. Transition table and flowchart representation implementations will be very similar.

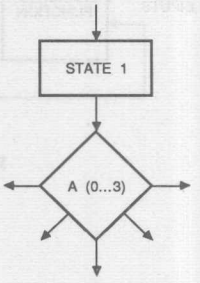
State Machine Types: Mealy & Moore

With the state machine representation clarified, we can now return to the generic sequencer model of Figure 1, which has been labelled (Figure 12) to show the present state (PS), next state (NS) and output (OB, OA). This will illustrate how Mealy and Moore machines are implemented with most sequencer devices which provide a single combinatorial logic array for both next state and output decode functions. There are four ways of using the sequencer, two of which implement Moore machines and two Mealy. First, let us look at the Mealy forms.

The standard Mealy form is shown in Figure 13, where the signals are labelled as in Figure 12 to indicate which registers and outputs are used. The register outputs PS are fed back into the array and define the present state. The combinatorial logic implements the transition function, which produces the next state flip-flop inputs NS, and the output function, which produces the machine output OB. This is the asynchronous Mealy form.



a. Testing Expressions

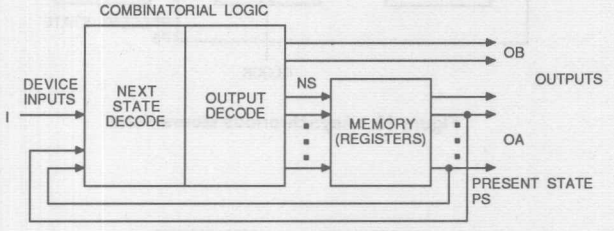


b. Multiway Branch

407 11a

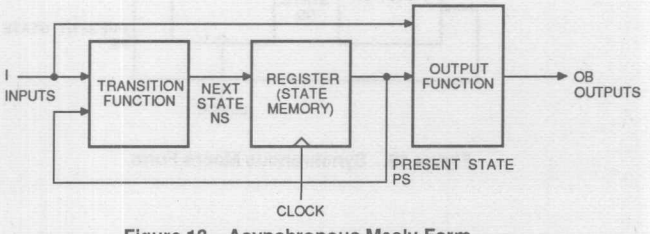
407 11b

Figure 11. Using Flowcharts



407 11

Figure 12. Generic Model of an FSM



407 12

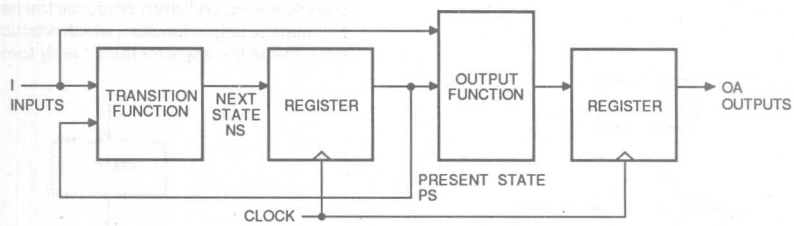
Figure 13. Asynchronous Mealy Form

An alternative Mealy form is shown in Figure 14. Here the outputs are passed through an extra output register (OA) and thus do not respond immediately to input changes. This is the synchronous Mealy form.

The standard Moore form is given in Figure 15. Here the outputs OB depend only on the present state PS. This is the asynchronous Moore form. The synchronous Moore form is shown in Figure 16. In this case the combinatorial logic can be assumed to be the unity function. The outputs (OB) can be generated directly along with the present state (PS). Although these forms have been described separately, a single sequencer is able to realize a machine which combines them, provided that the required paths exist in the device.

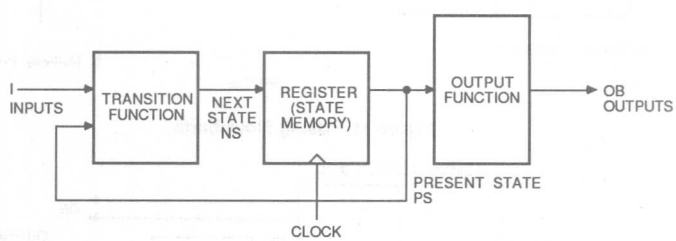
In the synchronous Moore form, the outputs occur in the state in which they are named in the state transition diagram. Similarly in the asynchronous Mealy and Moore forms the outputs occur in the state in which they are named, although delayed a little by the propagation delay of the output decoder. This is because they are combinatorial functions of the state (and inputs in the Mealy case).

However, the synchronous Mealy machine is different. Here an output does not appear in the state in which it is named, since it goes into another register first. It appears when the machine is in the next state, and is thus delayed by one clock cycle. The state diagram in Figure 17 illustrates all of the possibilities on a state transition diagram.



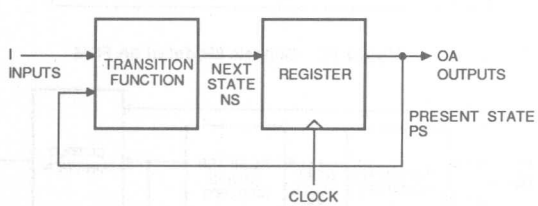
407 13

Figure 14. Synchronous Mealy Form



407 14

Figure 15. Asynchronous Moore Form



407 15

Figure 16. Synchronous Moore Form

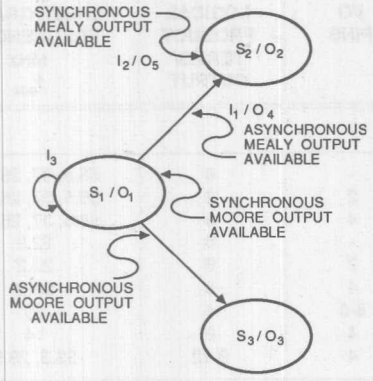


Figure 17. State Diagram Labelling for Different Output Types

As a matter of notation, Moore outputs are often placed within the state bubble and Mealy outputs are placed next to the path or arrow which activates them.

The relationship of Mealy and Moore, synchronous and asynchronous outputs to the states is shown in Figure 18.

Device Selection Considerations

Architecturally, the state machine devices can be divided into three categories:

- Logic-based devices
- Memory-based devices
- Instruction-based devices

Logic-based devices include the PAL and PLS devices. These devices use the sum-of-products logic array to implement the transition and output functions. The memory-based devices like PROSE implement the transition and output functions using a PROM or RAM array. Conceptually this is no different from the logic-based devices, since the memory can be viewed as special logic. The instruction-based sequencers (Am29PL141) offer hardwired instructions and fixed logic blocks to provide the transition function logic with enhanced capabilities like subroutines. Functionally all three types of devices work similarly performing two basic functions: control sequencing and output generation.

There are three major criteria for selecting the correct state machine device for a design:

- Number of inputs/outputs
  - I/O flexibility
  - Number of output registers
- Speed
- Intelligence/functionality
  - Number of product terms
  - Type of flip-flops
  - Number of state registers
  - Number of PROM locations (memory-based sequencers)

Number of I/Os

The number of inputs, outputs and I/O pins determines the signals which can be sampled or generated by a state machine. Figure 19 lists the devices offered for state machine designs, and shows the number of inputs, outputs and I/O pins available on each device.

Timing and Speed

The timing considerations for sequencer design are similar to those for registered logic design (page 2-64). A system clock

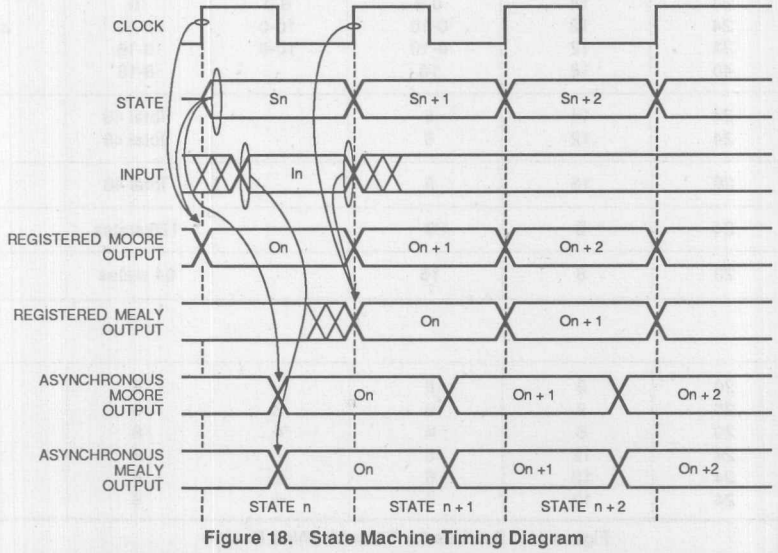


Figure 18. State Machine Timing Diagram

	NUMBER OF PINS	INPUT PINS	REGISTERED OUTPUT PINS	I/O PINS	PRODUCT TERMS/ OUTPUT	FREQUENCY IN MHz f_{MAX}
TTL						
16R8	20	8	8	-	8	55.5, 37, 25, 16
16R6	20	8	6	2	8	55.5, 37, 25, 16
16R4	20	8	4	4	8	55.5, 37, 25, 16
16RP8	20	8	8	-	8	22.2
16RP6	20	8	6	2	8	22.2
16RP4	20	8	4	4	8	22.2
16RA8	20	8	0-8	8-0	4	20
16X4	20	8	4	4	8	14
23S8	20	9	4	4	8-12	33.3, 28.5
20R8	24	12	8	-	8	37, 25, 16
20R6	24	12	6	2	8	37, 25, 16
20R4	24	12	4	4	8	37, 25, 16
20X10	24	10	10	-	4	22.2
20X8	24	10	8	2	4	22.2
20X4	24	10	4	6	4	22.2
20RP10	24	10	10	-	8	37, 25
20RP8	24	10	8	2	8	37, 25
20RP6	24	10	6	4	8	37, 25
20RP4	24	10	4	6	8	37, 25
20XRP10	24	10	10	-	8	30, 22.2, 14
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
20RS10	24	10	10	-	8-16	19.2
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
32R16	40	16	16	-	8-16	16
PLS167	24	14	6	-	Total 48	33
PLS168	24	12	8	-	Total 48	33
PLS105	28	16	8	-	Total 48	37
PMS14R21	24	8	8	-	128 states	25
29PL141	28	8	16	-	64 states	20
CMOS						
C16R8	20	8	8	-	8	28.5
C16R6	20	8	6	2	8	28.5
C16R4	20	8	4	4	8	28.5
C20R8	24	12	8	-	8	20, 15.3
C20R6	24	12	6	2	8	20, 15.3
C20R4	24	12	4	4	8	20, 15.3

Figure 19. Devices for State Machine Design

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
10H20EV/EG8	24	12	0-8	8-0	8-12	125
10020EV/EG8	24	12	0-8	8-0	8-12	125

Figure 19. Devices for State Machine Design (Cont'd.)

cycle forms the basic kernel for evaluating control function behavior. For the most part, all input and output functions are specified in relationship to the positive edge. Registered outputs are available after a period of time t_{CLK}, the clock-to-output propagation delay. Asynchronous outputs require an additional propagation delay (t_{PD}) before they are valid.

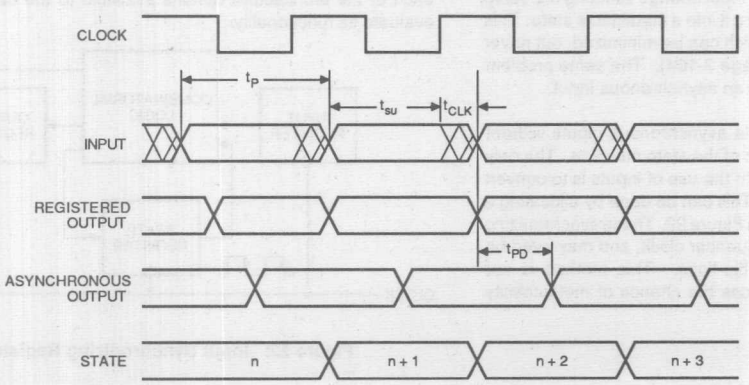
For the circuit to operate reliably, all of the flip-flop inputs must be stable at the flip-flop no later than the minimum set-up time (t_{su}) of the flip-flops before the next active clock edge. If one of the inputs changes after this threshold, then the next state or synchronous output could be stored incorrectly; the circuit may even malfunction. To avoid this, the clock period (t_p) must be greater than the sum of the set-up time of the flip-flops and the clock to output time ($t_{su} + t_{CLK}$). This determines the minimum clock period and hence the maximum clock frequency, f_{MAX}, of the circuit. Metastability and erroneous system operation may occur if these specifications are violated.

The timing relationships are shown in Figure 20. In each cycle there are two regions: the stable region, when all signals are steady, and the transition region, when the machine is changing state and signals are unstable. The active clock edge causes the flip-flops to load the value of the new state which has been set up at their inputs. At a time after this, the present state and output flip-flop outputs will start to change to their new values. After a time has elapsed, the slowest flip-flop output will be stable at its new value. Ignoring input changes for the moment, the changes in the state register cause the combinatorial logic to start generating new values for the asynchronous outputs and the inputs to the flip-flops. If the propagation delay of the logic is t_{PD}, then the stable period will start at a time equal to the sum of the maximum values of t_{CLK} and t_{PD}.

Figure 19 also shows the maximum operating frequency of the devices. ECL-based PAL sequencers can implement simple state machines from 60 MHz to above 100 MHz. Conventional TTL PAL sequencers can now reach speeds of 55 MHz. The new PLS devices operate at 37 MHz. Finally, PROSE sequencers provide 25 MHz operation, and the Am29PL141 can run at 20 MHz. The design engineer usually selects the simplest device that provides the desired level of performance.

Asynchronous Inputs

The timing of the inputs to a synchronous state machine are often beyond the control of the designer and may be random, such as sensor or keyboard inputs, or they may come from another



407 18

Figure 20. Timing Diagram for Maximum Operating Frequency

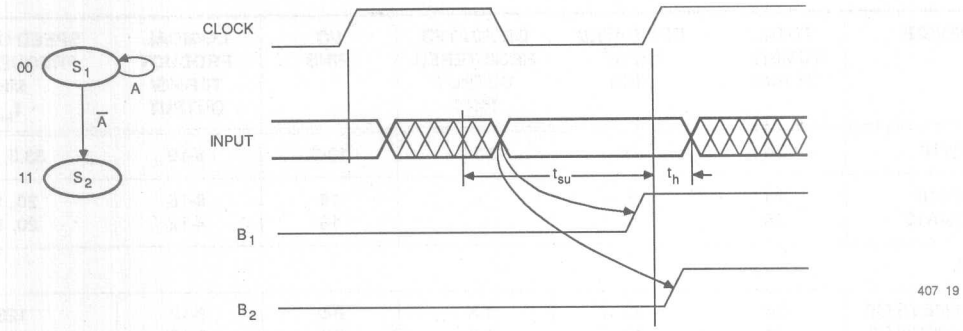


Figure 21. Asynchronous Input Causing Race

synchronous system that has an unrelated clock. In either case no assumptions can be made about the times when inputs can or cannot arrive. This fact causes reliability problems that cannot be completely eliminated, but only reduced to acceptable levels.

Figure 21 shows two possible transitions from state "S1" (code 00) either back to itself, or to state "S2" (code 11). Which transition is taken depends on input variable "A" which is asynchronous to the clock. The transition function logic for both state bits B1 and B2 includes this input. The input A can appear in any part of the clock cycle. For the flip-flops to function correctly, the logic for B1 and B2 must stabilize correctly before the clock. The input should be stable in a window t_{su} (setup time) before the clock and t_h (hold time) after the clock. If the input changes within this window, both the flip-flops may not switch, causing the sequence to jump to states 01 or 10, which are both undefined transitions. This type of erroneous behavior is called an input race.

A solution to this problem is to change the state assignment so that only one state variable depends on the asynchronous input. Thus the 11 code must be changed to 01 or 10. Now, with only one unsynchronized flip-flop input, either the input occurs in time to cause the transition, or it does not, in which case no transition occurs. In the case of a late input, the machine will respond to it one cycle later, provided that the input is of sufficient duration.

There is still the possibility of an input change violating the setup time of the internal flip-flop, driving it into a metastable state. This can produce system failures which can be minimized, but never eliminated (see Metastability, page 3-164). The same problem arises when outputs depend on an asynchronous input.

Very little can be done to handle asynchronous inputs without severely constraining the design of the state machine. The only way to have complete freedom in the use of inputs is to convert them into synchronous inputs. This can be done by allocating a flip-flop to each input as shown in Figure 22. These synchronizing flip-flops are clocked by the sequencer clock, and may even be the sequencer's own internal flip-flops. This method is not foolproof, but significantly reduces the chance of metastability occurring.

Functionality

The functionality of different devices is difficult to compare since different device architectures are available. The number of registers in a device determines the number of state combinations possible. However, all of the possible state combinations are not necessarily usable, since other device constraints may be reached. The number of registers do give an idea of the functionality achievable in a device. Other functionality measures include the number of product terms, type of flip-flop, or the number of PROM locations in a memory-based sequencer. One device may be stronger than another in one of these measures, but overall may be less useful due to other shortcomings. Choosing the best device involves both skill and experience.

A designer has a complete spectrum of devices with different architectures to choose from for a state machine design. These range from the PAL16R8D family of very high speed devices to the new PROSE PMS14R21 and instruction-based Am29PL141 high-functionality devices. The spectrum is completed by mid-range PAL devices including the PAL23S8, PAL22RX8A and PAL32VX10/A, with architectural features specifically designed for state machine designs, and the PLS devices designed exclusively for state machine implementation.

In order to give an idea of device functionality, we will consider each of the architecture options available to the designer and evaluate its functionality.

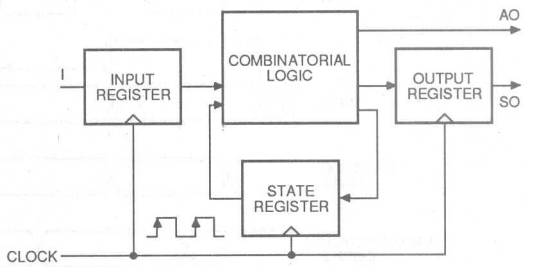


Figure 22. Input Synchronizing Register

PAL Devices as Sequencers

A vast majority of state machine designs are implemented with PAL devices. Early versions of software required the user to manually write the sum-of-products Boolean equations for using PAL devices. Second generation software allows one to specify the design in "state machine syntax," and handles the translation to sum-of-products logic automatically. PAL devices implement the output and transition functions in sum-of-products form via a user-programmable AND array and a fixed OR array.

PAL devices deliver the fastest speed of any sequencer and are ideally suited for simple control applications characterized by few input and output signals interacting within a dedicated controller in a sequential manner. The number of flip-flops in a PAL device range from 8 to 12, which offer potentially more than one thousand state values. Since some of the flip-flops are used for outputs, and the number of product terms is limited, the usable number of states is reduced drastically. Generally, up to about 35 states can be utilized.

PAL Device Flip-flops

PAL device based sequencers implement small state machine designs, which have a relatively large number of output transitions. Since the output registers change with most state transitions, they can be used simultaneously as state registers, once the state values are carefully selected. Most PAL devices are used for small state machines, and efficiently share the same register for output and state functions. High-functionality PAL device based sequencers provide dedicated buried state registers when sharing is difficult.

As a state machine traverses from one state to another, every output either makes a transition (changes logic level) or holds (stays at the same logic level). Small state machine designs require relatively more transitions and fewer holds. As designs get larger, state machines statistically require relatively fewer transitions and more holds.

Most PAL devices provide D-type output registers. D-type flip-flops use up product terms only for active transitions from logic LOW to HIGH level, and for holds for logic HIGH level only. J-K, S-R, and T-type flip-flops use up product terms for both LOW-to-HIGH and HIGH-to-LOW transitions, but eliminate hold terms. Thus D-type flip-flops are more efficient for small state machine designs. Some high-end PAL devices offer the capability of configuring the flip-flops as J-K, S-R or T-types, which are more efficient for large state machine designs since they require no hold terms.

Many examples of PAL-device-based sequencers can be found in system time base functions, special counters, interrupt controllers, and certain types of video display hardware.

PAL devices are produced in a variety of technologies for multiple applications, and provide a broad range of speed-power options. PAL devices which can be used for sequencer designs are listed in Figure 23. We will consider the following PAL devices in detail.

PAL10H/10020EV/EG8
PAL16R8D family
PAL23S8
PAL22RX8
PAL32VX10

DEVICE	TOTAL NUMBER OF PINS	DEDICATED INPUT PINS	DEDICATED REGISTERED OUTPUT PINS	I/O PINS	LOGICAL PRODUCT TERMS/ OUTPUT	SPEED GRADES FREQUENCY IN MHz f_{MAX}
TTL						
16R8	20	8	8	-	8	55.5, 37, 25, 16
16R6	20	8	6	2	8	55.5, 37, 25, 16
16R4	20	8	4	4	8	55.5, 37, 25, 16
16RP8	20	8	8	-	8	22.2
16RP6	20	8	6	2	8	22.2
16RP4	20	8	4	4	8	22.2
16RA8	20	8	0-8	8-0	4	20
16X4	20	8	4	4	8	14
23S8	20	9	4	4	8-12	33.3, 28.5
20R8	24	12	8	-	8	37, 25, 16
20R6	24	12	6	2	8	37, 25, 16
20R4	24	12	4	4	8	37, 25, 16
20X10	24	10	10	-	4	22.2
20X8	24	10	8	2	4	22.2
20X4	24	10	4	6	4	22.2

Figure 23. Table of PAL Devices for Sequencer Applications

	NUMBER OF PINS	INPUT PINS	REGISTERED OUTPUT PINS	PINS	PRODUCT TERMS/ OUTPUT	FREQUENCY IN MHz f_{MAX}
20RP10	24	10	10	-	8	37, 25
20RP8	24	10	8	2	8	37, 25
20RP6	24	10	6	4	8	37, 25
20RP4	24	10	4	6	8	37, 25
20XRP10	24	10	10	-	8	30, 22.2, 14
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
20RS10	24	10	10	-	8-16	19.2
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
20XRP10	24	10	10	-	8	30, 22.2, 14
20XRP8	24	10	8	2	8	30, 22.2, 14
20XRP6	24	10	6	4	8	30, 22.2, 14
20XRP4	24	10	4	6	8	30, 22.2, 14
20RS10	24	10	10	-	8-16	19.2
20RS8	24	10	8	2	8-16	19.2
20RS4	24	10	4	6	8-16	19.2
20RA10	24	10	0-10	10-0	4	33, 20
22RX8	24	14	0-8	8-0	8	28.5
22V10	24	12	0-10	10-0	8-16	40, 28.5, 18
32VX10	24	12	0-10	10-0	8-16	25, 22.2
32R16	40	16	16	-	8-16	16
CMOS						
C16R8	20	9	8	-	8	28.5
C16R6	20	9	6	2	8	28.5
C16R4	20	9	4	4	8	28.5
C20R8	24	13	8	-	8	20, 15.3
C20R6	24	13	6	2	8	20, 15.3
C20R4	24	13	4	4	8	20, 15.3
C22V10	24	12	0-10	10-0	8-16	33.3, 20
C29M16	24	5	-	16	8-16	20, 15
C29MA16	24	5	-	16	4-12	20, 15
ECL						
10H20EV/EG8	24	12	0-8	8-0	8-12	125
10020EV/EG8	24	12	0-8	8-0	8-12	125

Figure 23. Table of PAL Devices for Sequencer Applications (Cont'd.)

The ECL PAL10H/10020EV/EG8

At 125 MHz, the ECL PAL10H/10020EV/EG8 provides the highest speed for a state machine design. The PAL10H/10020EV/EG8 has eight outputs and 20 inputs. Half of the outputs have 8 product terms and half have 12 product terms. All of the 20EV8 outputs use D-type flip-flops, while the 20EG8 outputs are transparent latches. Two architectural fuses per output control the polarity of the logic and bypass the flip-flop for combinatorial asynchronous outputs. Two global product terms are present to Set and Reset all flip-flops. Maximum frequency of operation as a state machine is 125 MHz.

Figure 24 details the macrocell for the PAL10H/10020EV8.

The PAL16R8D Family

This is the high-speed end (55 MHz) of the TTL state machine design solutions. PAL16R8D family devices are available in three different versions (PAL16R8/6/4) with varying numbers of registered and combinatorial outputs and I/O pins for convenient design fitting. The PAL16R8 family provides 64 product terms (eight per output).

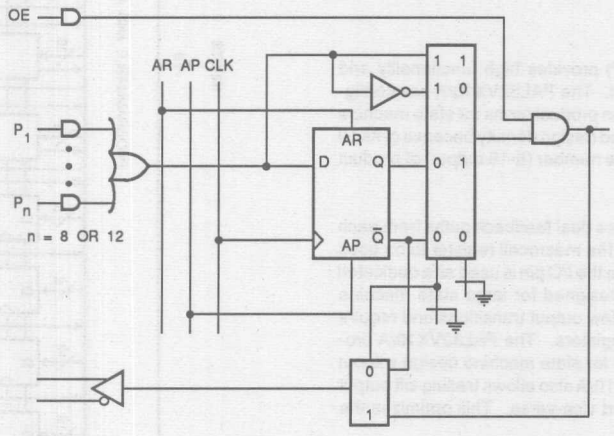
The software design options available for the PAL16R8D family are the traditional Boolean equations and the state machine syntax provided by PALASM 2 software.

The PAL23S8

At 33.3 MHz and in a 20-pin DIP, this device (Figure 25) provides the highest performance-to-board-space ratio. Designed for high-functionality state machines, this device offers eight output and six dedicated buried registers. It also offers four output macrocells with register bypass for asynchronous outputs. It has nine dedicated device inputs and four I/O pins giving a total of 23 array inputs. For improved state machine initialization, it also offers dedicated synchronous Preset and asynchronous Reset product terms. Pin-compatible with the PAL16R8 family, this device offers varied distribution of 145 product terms for large designs and optimal fit.

The software design options include Boolean equations and state machine syntax.

2



630 02

Figure 24. The PAL10H/10020EV8 Output Macrocell

The PAL22RX8A

The 24-pin PAL22RX8A (Figure 26) provides eight configurable input/output macrocells with register bypass for a user-programmable number of registered or combinatorial outputs and I/O pins. It also provides dedicated asynchronous reset and preset product terms for state machine initialization. The PAL22RX8A provides a product-term-controlled XOR gate with its sum-of-products logic. This allows users to configure individual macrocells with D, J-K, S-R or T-type flip-flops.

The requirements of transition and hold terms are dependent upon state selection, which varies for different applications. The PAL22RX8A provides both D-type and J-K type flip-flops, allowing the designer to select the one requiring fewer product terms.

The PAL22V10

The PAL22V10 provides output macros similar to the PAL22RX8A. It provides both programmable polarity and register bypass. In addition, it has varied product term distribution.

The PAL22V10 has an advantage of a large number of product terms. It provides a varied distribution of eight to sixteen product terms per output. It also provides ten output macrocells, allowing for larger designs.

The PAL32VX10/A

The PAL32VX10/A (Figure 27) provides high functionality and density for PAL device designs. The PAL32VX10/A has configurable flip-flops and initialization product terms for state machine designs. It offers unprecedented design density because of its 10 input/output macros and a large number (8-16/output) of product terms.

The PAL32VX10/A also provides dual feedback paths from each output macrocell. This allows the macrocell register to be used as a buried state register in case the I/O pin is used as a dedicated input. The PAL32VX10/A is designed for large state machine designs which have relatively few output transitions and require separate output and buried registers. The PAL32VX10/A provides up to 10 buried registers for state machine design without any I/O penalty. The PAL32VX10/A also allows trading-off output registers for state registers and vice-versa. This optimizes the use of device resources.

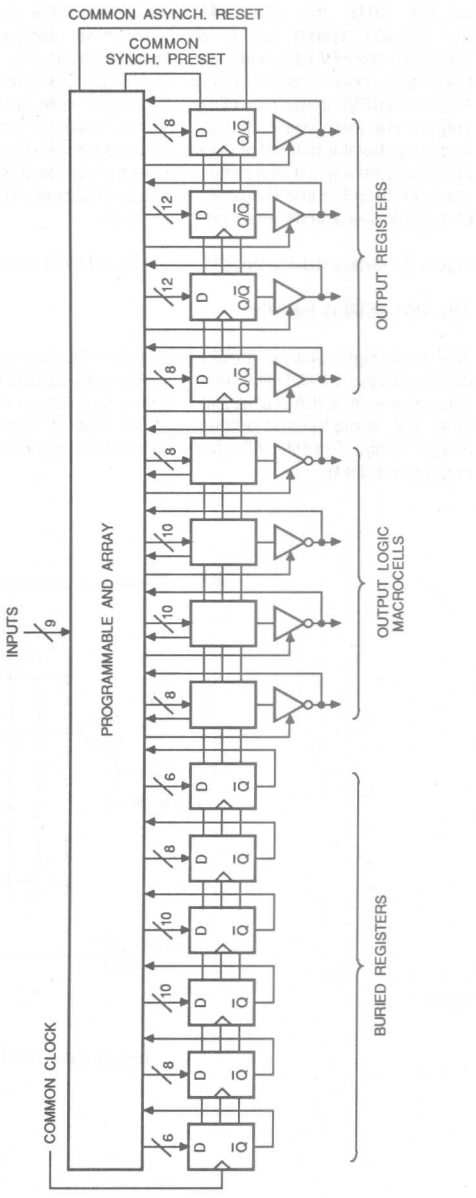
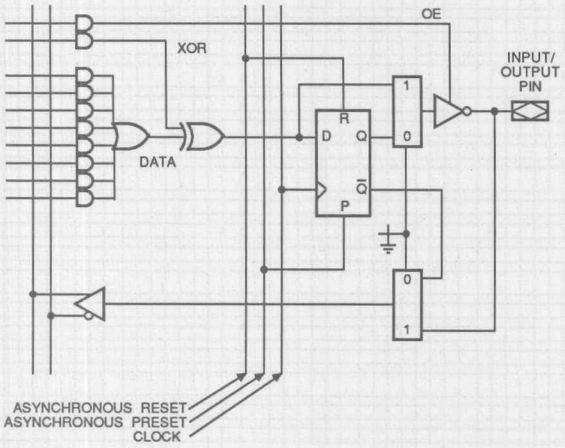


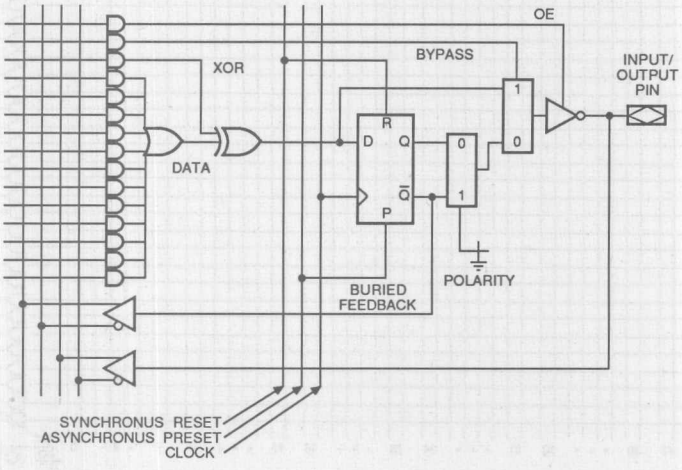
Figure 25. Block Diagram of PAL23S8

407 22



407 23

Figure 26. PAL22RX8A Output Macrocell



407 24

Figure 27. PAL32VX10/A Output Macrocell

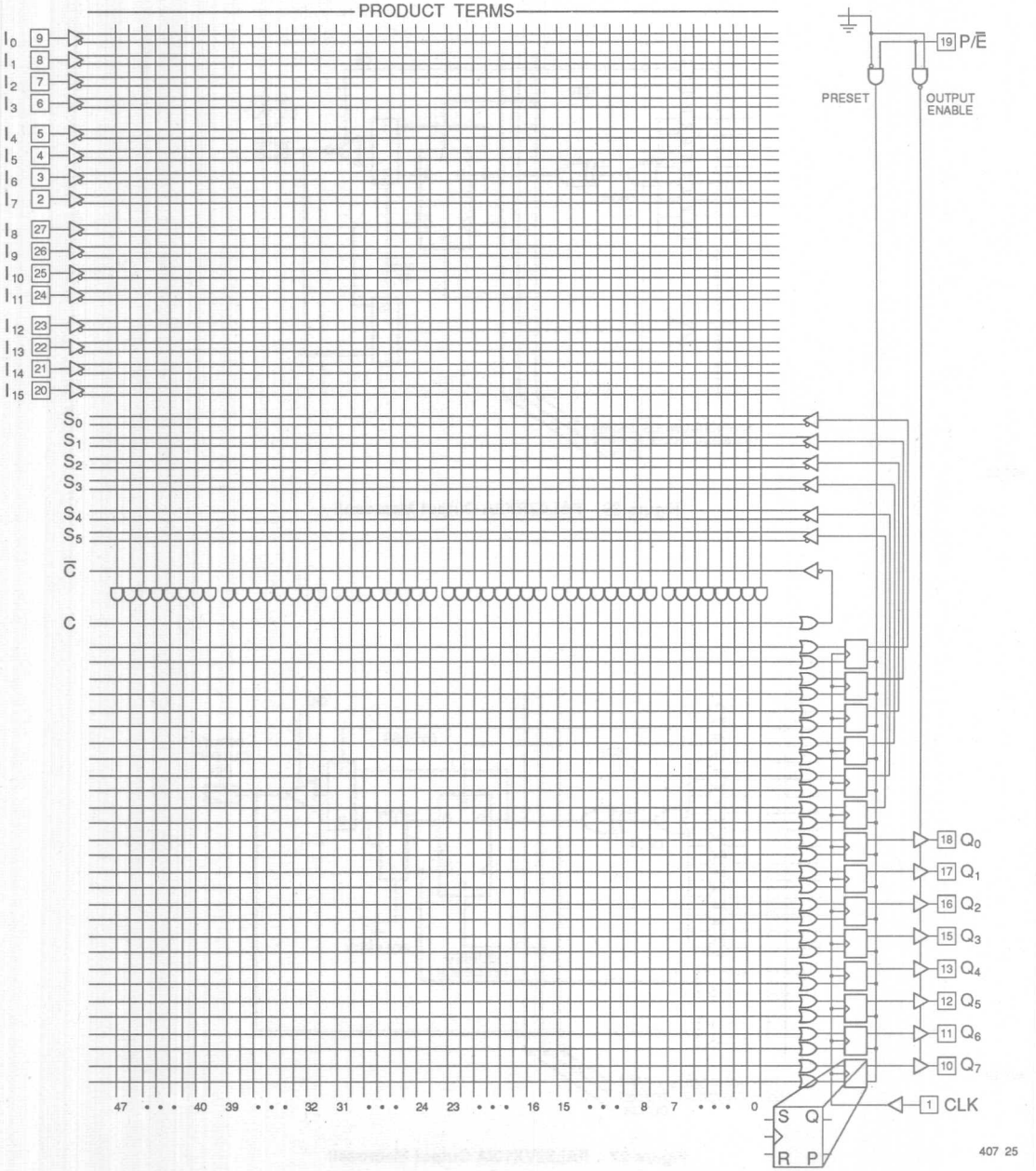


Figure 28. Architecture of a PLS105 Device

Programmable Logic Sequencers (PLS)

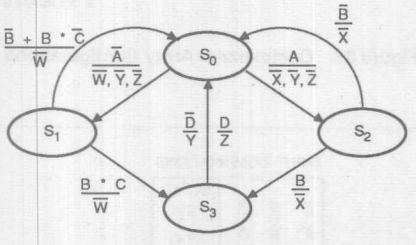
Another alternative for control logic is the PLS (Programmable Logic Sequencer) device (Figure 28). PLS devices have both programmable AND and programmable OR arrays. Forty-eight product terms are available, which can be shared between transition functions and output functions for all of the outputs, as dictated by the application needs. PLS devices also offer S-R flip-flops with buried registers for control sequencing. The flip-flops have a common clock line. The S-R type flip-flops are similar to the J-K type flip-flops discussed earlier, and require no hold terms. Consequently S-R type flip-flops are very efficient for large state machine designs. PLS devices can function as either as Moore machines or Mealy machines with a register on the output path.

With the improvements in design tools, it is possible to specify designs directly from the state diagram. However, one format which has been used often in the past is the PLS table. This table is primarily used where the design is entered manually. The table is also useful for learning about and experimenting with sequencer designs. Often, it is possible to manipulate sequencer programs using the table in ways which are not possible with some of the more rigid CAD languages.

The table has provision for input and output signal pin allocation and naming, and a line for each of the 48 device product terms. The entries used in each line are shown in Figure 29.

VALUE	AND ARRAY TERMS	OR ARRAY TERMS
	(input, present state)	(next state, output)
H	true input/state bit	Set Flip-flop
L	complement	Reset Flip-flop
-	don't care	no change

Figure 29. Symbols Used in a PLS Table



407 28a

Figure 30. Example State Machine

The table defines the output function and transition function. Both functions depend on the *present state* and *inputs*. Each possible link path between two states uses a product term and becomes a line in the PLS table. The collection of all link path lines is a complete definition of the combinatorial logic required. The following example illustrates the relationship between a state diagram and the PLS table.

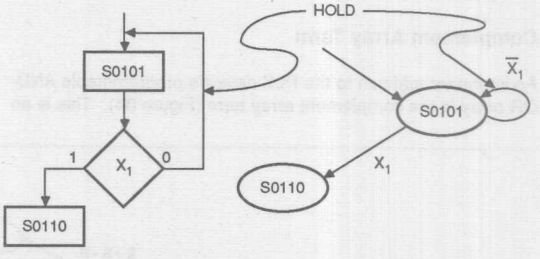
Let us look at an arbitrary function as shown in Figure 30. The inputs are A,B,C, and D; the state names are S0, S1, S2 and S3; and the outputs are W, X, Y and Z. To construct the PLS table, a state assignment is necessary. Selecting an arbitrary binary code for each state:

S0 = 00
S1 = 01
S2 = 10
S3 = 11

With the two state variables P and Q, the PLS table can be derived, and is shown in Figure 31.

PATH #	INPUTS	PRESENT STATE	NEXT STATE	OUTPUTS
	A B C D	P Q	P Q	W X Y Z
1	H - - -	L L	H L	L H L L
2	L - - -	L L	L H	H L L L
3	- H H -	L H	H H	L L L L
4	- L - -	L H	L L	L L L L
5	- H L -	L H	L L	L L L L
6	- H - -	H L	H H	L L L L
7	- L - -	H L	L L	L L L L
8	- - - H	H H	L L	L L L H
9	- - - L	H H	L L	L L H L

Figure 31. Example PLS Table



407 28b

Figure 32. "HOLD" Transitions

This is now a complete description of the combinational logic of the state machine. Every line in the table is a product term which is active when conditions for a transition to another state are satisfied. For this reason, the product terms in a PLS device are also known as *transition terms*. As the circuit moves from state to state, new transition terms become active as others become inactive.

A comment may be appended to each term to aid documentation of the design.

"Hold" Terms

The flip-flops used in PLS devices are the S-R type. As discussed earlier, S-R flip-flops do not require any product terms for holds and require product terms only for state transition or output generation (Figure 32).

This may be seen in the two lines of the PLS table corresponding to the two link paths shown, in Figure 33.

INPUT X1	PRESENT STATE	NEXT STATE
L	L H L H	- - - -
H	L H L H	- - H L

Figure 33. Hold Transitions in a PLS Table

The first path, corresponding to the first line in Figure 33, is a transition back to the same state. No change is thus required in any of the state variables. '-' entries can thus be used. The first line has only '-' entries in the OR Array part of the table; the transition term is thus not connected and this line can be eliminated altogether. This is a logic minimization which can be done immediately when converting the flowchart or state diagram into a PLS table. The second path is to a different state, "0110". This does require a term since some state variables have to change. Also, note that "hold" state transitions are only free if no output changes are required.

Complement Array Term

An important addition to the PLS device's programmable AND-OR array is the complement array term (Figure 34). This is an extra OR line which is first complemented and then fed back as an additional input into the AND array. Figure 35 shows the three possible configurations of the array at each transition term. The symbols shown are used in an extra truth table column for configuring the complement array.

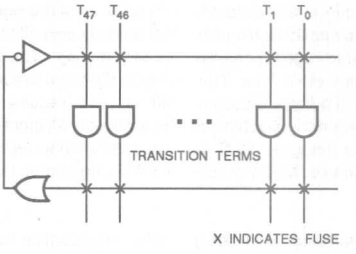


Figure 34. Connections of the Complement Array Term

Proper usage of the complement array can lead to considerable savings in product terms. This can be seen in the typical use of the complement array in the implementation of default or "ELSE" transitions. Figure 36 shows a set of four transitions from state SA, three of which are enabled by one of three combinations of inputs A, B, and C, causing transitions to states S0, S1 and S2. All other combinations cause a transition to state SX.

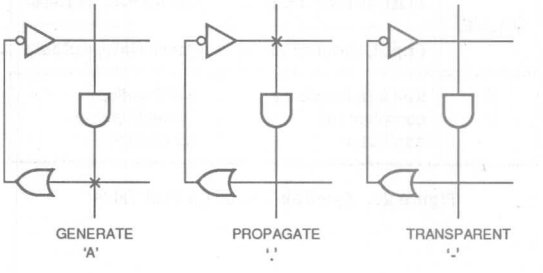
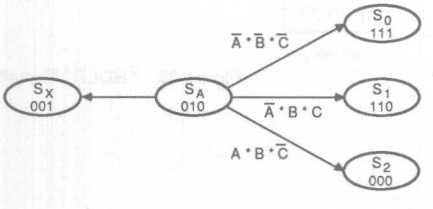


Figure 35. Complement Array Configurations



INPUT COMBINATIONS

$\overline{A}$	$\overline{B}$	$\overline{C}$	GO TO
$\overline{A}$	$\overline{B}$	C	S_0
$\overline{A}$	B	$\overline{C}$	S_X
$\overline{A}$	B	C	S_X
A	$\overline{B}$	$\overline{C}$	S_1
A	$\overline{B}$	C	S_X
A	B	$\overline{C}$	S_X
A	B	C	S_2

407 28

Figure 36. Transitions with a Default

Three product terms are required for transition to states S0, S1 and S2 which encode the three different conditions. For the default transition to state SX, the remaining five condition combinations of A, B and C would have to be encoded, requiring five more product terms (Figure 37). But these terms can be thought of as "not the transitions to S0 or S1 or S2". By using the complement array (which provides this NOT signal), only one more product term is required for the default transition (Figure 38).

The last five terms have been compressed into one. If none of the three defined input combinations is true in state SA, the inputs to the complement array will all be zeros, and the complement array output will be a one. This activates the fourth transition term, causing a transition to SX (Figure 39).

The same complement array term can be used in other states too, since the propagated signal is ANDed with the present state code. Therefore all defaults can be realized with one term per state. PLS devices can be an effective solution in state- and branch-intensive applications, typically found in a variety of protocol controllers, waveform generators, and sequence detectors. In such applications, complex arrangements of multiple branches may be present. Multi-step control functions may include many loops of dozens of steps to count packets of data or input events as they occur.

PLS devices can be characterized by the number of inputs, number of product terms, number of flip-flops and number of outputs. The MMI PLS family currently has three members; these are listed in Figure 40.

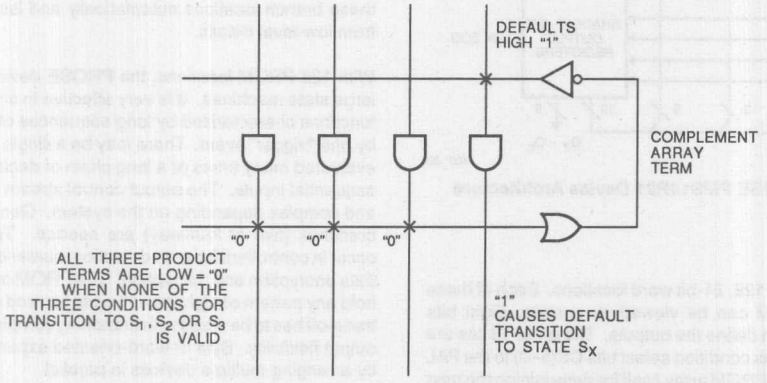
Software for PLS devices can usually accept input descriptions in three formats: truth tables, Boolean equations and state machine syntax. The first two require the user to work out the explicit logic assignment for all terms and code the detailed fuse states programmed or unprogrammed. The new state machine syntax allows design in a high-level syntax, and is easier to use.

Cn	A	B	C	PRESENT STATE	NEXT STATE	COMMENT
-	L	L	L	L H L	H - H	to S0
-	L	H	H	L H L	H - -	to S1
-	H	H	L	L H L	- L -	to S2
-	L	L	H	L H L	- L H	path to SX
-	L	H	L	L H L	- L H	
-	H	L	L	L H L	- L H	
-	H	L	H	L H L	- L H	
-	H	H	H	L H L	- L H	

Figure 37. Product Terms Required for Transition to State SX

Cn	A	B	C	PRESENT STATE	NEXT STATE	COMMENT
A	L	L	L	LHL	H-H	
A	L	H	H	LHL	H- -	
A	H	H	L	LHL	-L-	
•	-	-	-	LHL	-LH	path to state SX

Figure 38. Default Transition Using Complement Array Term



407 29

Figure 39. Transitions Using Complement Array Term

	NUMBER OF PINS	INPUT PINS	REGISTERED OUTPUT PINS	PINS	PRODUCT TERMS/ OUTPUT	FREQUENCY IN MHz f_{MAX}
TTL						
PLS167	24	14	6	-	Total 48	33
PLS168	24	12	8	-	Total 48	33
PLS105	28	16	8	-	Total 48	37

Figure 40. Table of PLS Devices

PROSE Sequencer (PMS14R21)

The PROSE (PMS14R21 PROgrammable SEquencer) is a revolutionary architecture optimized for very high functionality state machine designs. It combines a PROM and a PAL array on a single device, utilizing the efficiencies of both for state machine design.

The PROSE is a high-speed, 14-input, 8-output state machine. It consists of a 128x21 PROM array preceded by a 14H2 PAL array (see Figure 41). The PAL array is efficient for performing logic functions on a large number of input conditions, while the PROM array is optimal for implementing a large number of product terms (decision branches) and states. The combination allows a very efficient implementation of a state machine.

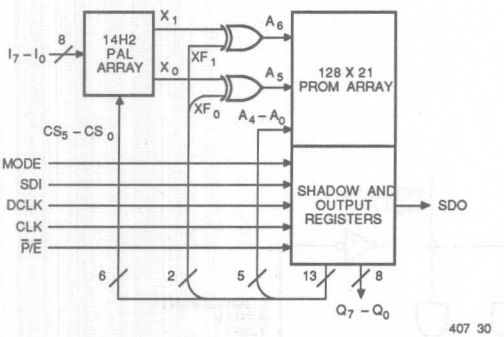


Figure 41. PROSE PMS14R21 Device Architecture

PROM Operation

The PROM consists of 128, 21-bit word locations. Each of these locations in the PROM can be viewed as a state. Eight bits Q(7–0) at each location define the outputs. The other 13 bits are used as feedback: six as condition select bits CS(5–0) to the PAL array, and seven to the PROM array itself for determining the next location to be addressed. Five of these seven constitute the low-order address bits A(4–0) for the next state. The remaining 2 bits are XF(1–0), inputs to the exclusive-OR gates which determine address bits A6 and A5 in conjunction with PAL output signals X1 and X0. Effectively, the seven-bit address of every PROM location is generated by seven bits of the PROM data and constitutes the present state. The next PROM location is the one whose address is stored as data in the present location. Eight of the PROM data bits are used as outputs.

PAL Array Operation

The PAL14H2 has 14 complementary inputs and 2 active-HIGH outputs (X1 and X0). Each of the 2 outputs is a sum of eight product terms. Eight of the inputs, I(7–0), are from an external source, and are used for encoding conditions for branch selection. Six inputs, CS(5–0), are from internal feedback from the PROM, and are used for selection of the conditions that determine branching when decisions are involved in state transitions.

Just as PROMs are efficient for direct control sequencing, PAL arrays are very efficient for encoding input conditions. The function of the PAL14H2 in the PROSE device is to encode all eight test input conditions along with the state information. Based upon these test conditions being true it inverts the polarity of the two most significant PROM address bits. This causes the PROM to address a different location (state), which is equivalent to conditional branching. The PALASM 2 design software selects these branch locations automatically and isolates the designer from low-level details.

With 128 PROM locations, the PROSE device can be used for large state machines. It is very effective in single-thread control functions characterized by long sequences of events generated by one "trigger" event. There may be a single repetitive decision evaluated many times or a long chain of decisions evaluated for sequential inputs. The output control stream may be quite wide and complex depending on the system. Generally, only narrow branches (two or four-way) are needed. Typical applications occur in controllers for error detection, numerical processing, and data encryption and decryption. The PROM output structure can hold any pattern of eight-bit values to be read out as needed. No trade-off has to be made with branching complexity to provide this output flexibility. Byte or word-oriented expansion can be made by arranging multiple devices in parallel.

PROSE state machines are fully supported by PALASM 2 software. A complete design example of a PROSE-based state machine is included on page 2-122.

Fuse Programmable Controller (Am29PL141)

The Am29PL141 is a high-performance, single-chip, fuse-programmable controller (FPC). This chip is designed to allow the implementation of complex state machines and controllers by programming the appropriate sequence of microinstructions in the on-chip microprogram memory.

Large state machines require the capability of nested routines. Nested state machines are control sequences that may be invoked by one or more control functions, but when invoked, suspend the "calling" control sequence. Following completion of the low-level routine, the higher-level "calling" sequence resumes where it left off. The nested task may have modified part of the machine environment or the data processed by the system. The Am29PL141 provides stacks which allow nested subroutines.

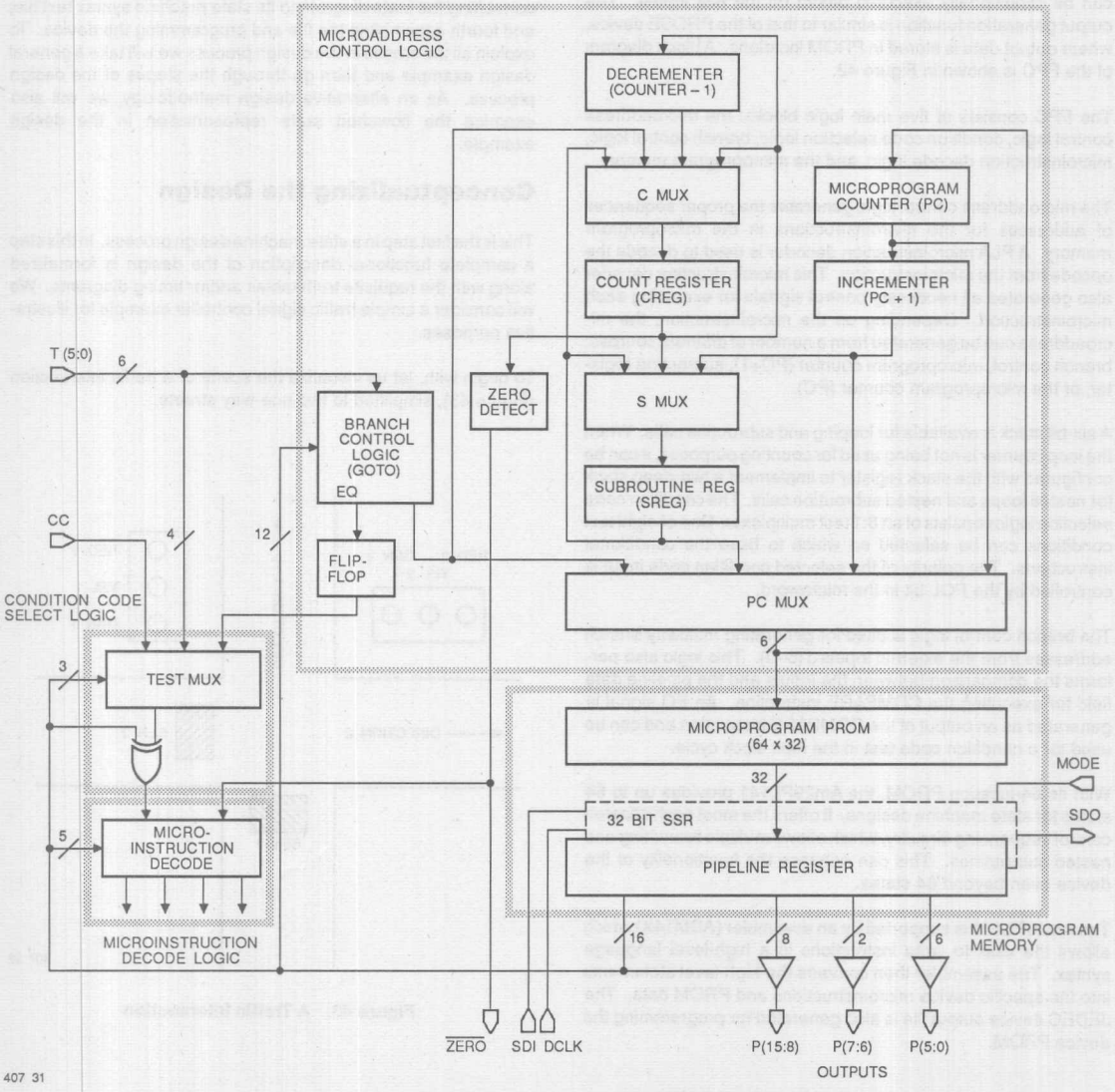


Figure 42. Architecture of Am29PL141

With its on-chip intelligent microprogram address sequencer (Figure 42), high-speed 64 x 32-bit PROM-based microprogram memory, on-chip pipeline register, and an on-chip diagnostics serial shadow (SSR) register, this chip offers two major advantages: fast operation, and testability. This device is available with a 20-MHz clock rate (50-ns cycle time) in a 28-pin dual-in-line package.

A microprogram address sequencer is the heart of the FPC, and performs all control sequencing functions. The Am29PL141 has 29 high-level microinstructions which include jumps, loops, subroutine calls, and multiway branching. These microinstructions can be conditionally executed based on the test inputs. The output generation function is similar to that of the PROSE device, where output data is stored in PROM locations. A block diagram of the FPC is shown in Figure 42.

The FPC consists of five main logic blocks: the microaddress control logic, condition code selection logic, branch control logic, microinstruction decode logic, and the microprogram memory.

The microaddress control logic generates the proper sequences of addresses for the microinstructions in the microprogram memory. A PLA microinstruction decoder is used to decode the opcode from the microinstruction. This microinstruction decoder also generates all necessary control signals for executing each microinstruction. Depending on the microinstruction, the microaddress can be generated from a number of different sources: branch control, microprogram counter (PC+1), subroutine register, or the microprogram counter (PC).

A six-bit stack is available for looping and subroutine calls. When the loop counter is not being used for counting purposes, it can be configured with the stack register to implement a two-deep stack for nested loops and nested subroutine calls. The condition code selection logic consists of an 8:1 test multiplexer. One of eight test conditions can be selected on which to base the conditional instructions. The polarity of the selected condition code input is controlled by the POL bit in the microword.

The branch control logic is used for generating multiway branch addresses from the external inputs T(5–0). This logic also performs the comparison between the inputs and the pipeline data field for executing the COMPARE instruction. An EQ signal is generated as an output of the COMPARE instruction and can be used for a condition code test in the next clock cycle.

With its 64-location PROM, the Am29PL141 provides up to 64 states for state machine designs. It offers the most sophisticated control sequencing circuitry, which allows multiple branching and nested subroutines. This can enhance the functionality of the device even beyond 64 states.

The Am29PL141 is supported by an assembler (ASM14X) which allows the user to write instructions in a high-level language syntax. The assembler then converts the high-level statements into the specific device microinstructions and PROM data. The JEDEC device output file is also generated for programming the device PROM.

State Machine Design Tutorial

The following discussion is a tutorial on state machine design methodology. In this tutorial we will use the PROSE PMS14R21 device and explain the state machine design process. The PROSE PMS14R21 is supported by the state machine syntax of PALASM 2 design software. This software syntax is identical to and can be used for programming other PAL and PLS devices.

There are four stages in a state machine design. The first requires conceptualizing the design and selecting the correct device; second converting it to its state diagram representation; third converting the state diagram to its state machine syntax text file; and fourth assembling the file and programming the device. To explain all the stages in the design process we will take a general design example and then go through the stages of the design process. As an alternative design methodology, we will also examine the flowchart state representation in the design example.

Conceptualizing the Design

This is the first step in a state machine design process. In this step a complete functional description of the design is formalized along with the requisite truth tables and/or timing diagrams. We will consider a simple traffic signal controller example for illustrative purposes.

To begin with, let us visualize the scene of a traffic intersection (Figure 43), simplified to two one-way streets.

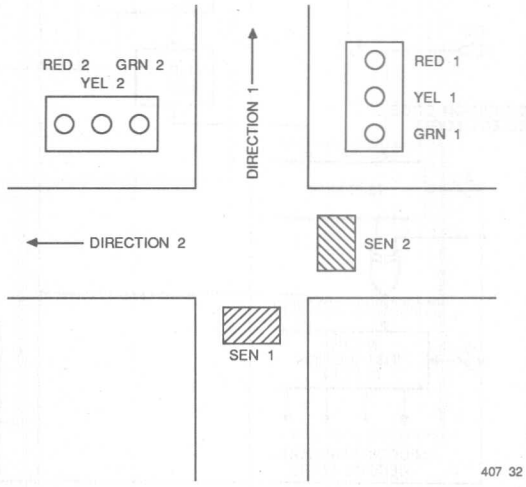


Figure 43. A Traffic Intersection

The traffic intersection shows two one-way streets: one in direction 1 and the other in direction 2. Each direction has a signal consisting of red, yellow, and green lamps. These lamps are activated with appropriately-named active-HIGH signals (RED_1, YEL_1, GRN_1, RED_2, YEL_2, GRN_2). The signals are generated by the state machine controller. Also, each direction has a sensor which provides an active-HIGH signal (SEN_1, SEN_2) that indicates the presence of a vehicle. The controller has to manage this intersection, with the sensors as inputs and the lamps as outputs.

The assertion of SEN_1 or SEN_2 signals a request for a green light for traffic in the corresponding direction. Once SEN_1 is HIGH and SEN_2 is LOW, indicating traffic in direction 1, the GRN_1 light should be on at all times. Similarly, when SEN_2 is HIGH and SEN_1 is LOW, the GRN_2 light should be on at all times. When SEN_1 and SEN_2 are both HIGH, indicating traffic in both directions, the traffic controller should cycle, allowing equal periods of green signals (GRN_1 and GRN_2) for both directions. This cycling is also done when SEN_1 and SEN_2 are both LOW, indicating no traffic.

SEN_2	SEN_1	
L	H	Allow traffic in direction 1: RED_2, GRN_1 = H
H	L	Allow traffic in direction 2: RED_1, GRN_2 = H
H	H	Cycle with equal durations in both directions
L	L	Cycle with equal durations in both directions

Figure 44. Table for Traffic Flow Direction

Whenever the signals change from direction 1 to direction 2, the appropriate yellow light (YEL_1 or YEL_2) is turned on for the transition duration. This allows time for the traffic in the intersection to pass, before allowing traffic in the other direction. The timing diagram is shown in Figure 45. When both sensors indicate the presence of traffic (SEN_1, SEN_2 = H), extra time is allowed for traffic to pass, before the signals change the direction. The extra time is also allowed when there is no traffic in both directions (SEN_1, SEN_2 = L). The timing diagram for this is shown in Figure 46. Finally, the traffic signal controller shown includes the system clock (CLK), and an initialize or reset signal (INIT). INIT drives the controller to an initial state.

Before we proceed to the next step, we must select a device which will implement this design.

In the previous section we found that the basic selection of a state machine controller device depends upon the speed and I/O considerations. The PROSE PMS14R21 has eight inputs and eight outputs in addition to the initialization input signal PRESET, which are sufficient for this design. Another selection criterion is the intelligence of the PROSE device. The device should be able to implement all of the states required by the design. The PROSE device is the only 24-pin SKINNYDIP device which has the capacity for up to 128 states. Later we will see that these are sufficient for implementing our design.

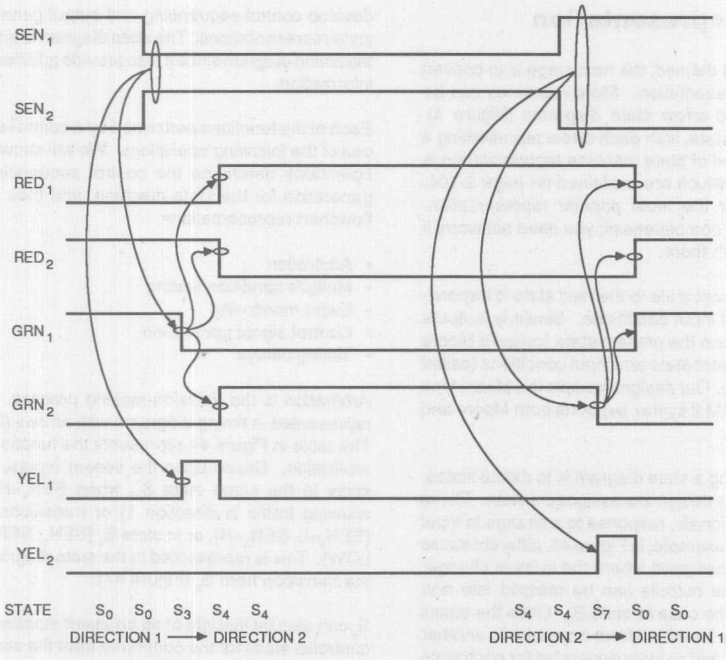


Figure 45. Fast Traffic Flow Transition

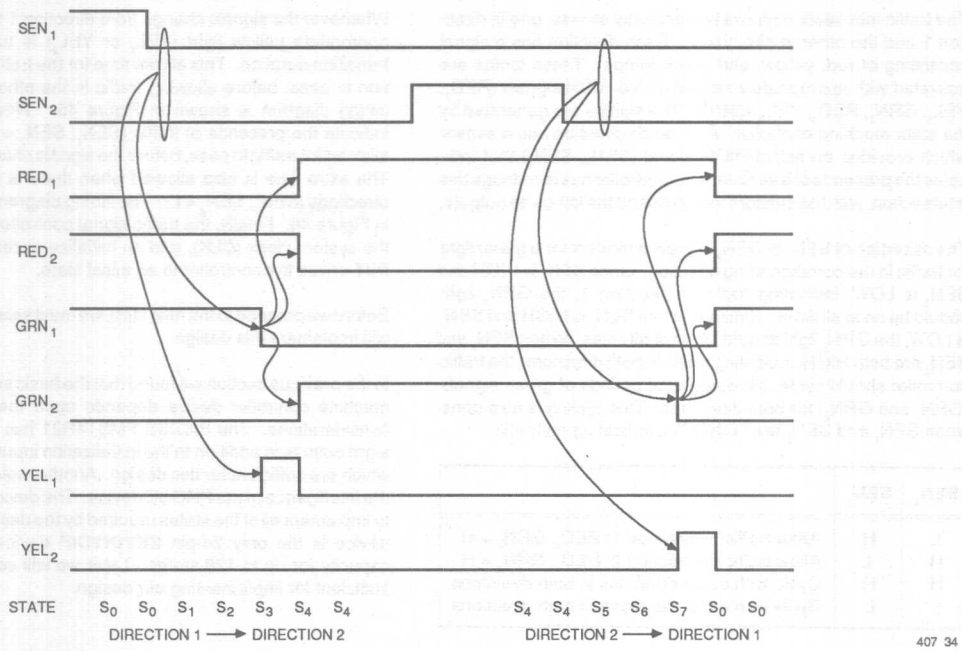


Figure 46. Slow Traffic Flow Transition

State Machine Representation

Once the design problem is defined, the next stage is to convert it to its state machine representation. State machines can be represented by bubble-and-arrow state diagrams (Figure 4). Each bubble represents a state, with each arrow representing a transition. A second method of state machine representation is with flow charts, details of which are explained on page 2-104. State diagrams are by far the most popular representation. Flowcharts are included for completeness; you need not worry if you are not comfortable with them.

The transition from the present state to the next state is dependent upon present state and input conditions. Similarly, outputs can be generated based upon the present state (called a Moore type state machine), or present state and input conditions (called a Mealy type state machine). Our design example is a Moore type state machine. The PALASM 2 syntax supports both Moore and Mealy type state machines.

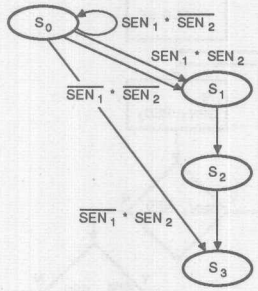
The first step for constructing a state diagram is to define states. All of the specific events in a design are assigned a state. These include: change of output signals, response to a change in input signals, or time delays. For example, in Figure 45, different states (S_0, S_3 and S_4) have been assigned where the outputs change. Often states with the same outputs can be merged into one unique state, as shown in the case of state S_0. Once the states are defined as bubbles, the transition from one state to another (called control sequencing) and outputs generated for each state (output generation) are defined. The task is to use the timing diagram along with the functional description of the design to develop control sequencing and output generation functions for state representations. The state diagram is an improvement over the timing diagram since it also provide additional branch decision information.

Each of the functions performed by a controller can be viewed as one of the following operations. We will show how each of these operations determine the control sequencing and the output generation for the state machine, and their state diagram and flowchart representations.

- Arbitration
- Multiple condition testing
- Event monitoring
- Control signal generation
- Timing delays

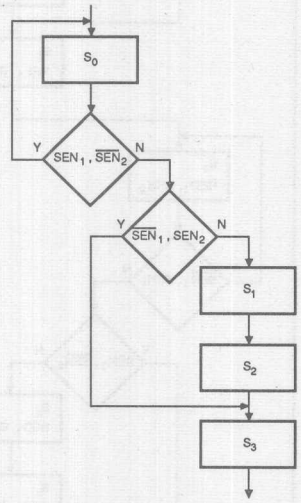
Arbitration is the decision-making process. It is inadequately represented in timing diagrams with arrows (Figures 45 and 46). The table in Figure 44 represents the functional requirement for arbitration. Based upon the sensor inputs, the state machine stays in the same state S_0, when SEN_1=H and SEN_2=L (for allowing traffic in direction 1) or transitions either to state S_3 (SEN_1=L, SEN_2=H), or to state S_1 (SEN_1, SEN_2 are both HIGH or LOW). This is represented in the state diagram and flowchart as the transition from S_0 (Figure 47).

S_0 can also be thought of as an *event monitoring* state where the controller waits for the command from the sensors. Based upon the two bits SEN_0 and SEN_1, the state machine moves to different next states. This is also an example of *multiple condition testing.*



a. State Diagram

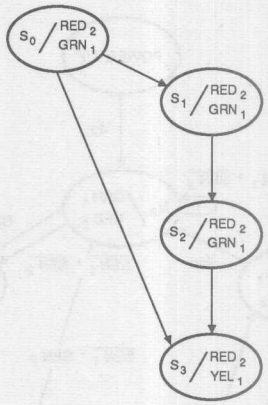
407 37



407 42a

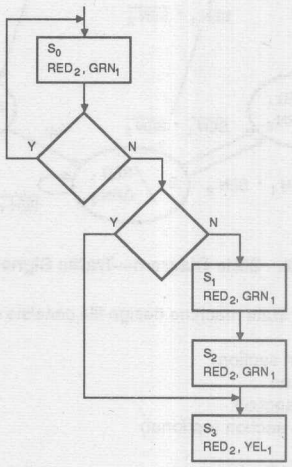
b. Flowchart

Figure 47. Conditional State Transitions



407 36

a. State Diagram



407 41a

b. Flowchart

Figure 48. Output Assignment to States

An example of *control signal generation* (Figure 48) is signal RED_2 which is used to turn on the red light for direction 1. This control signal generation example requires assigning output signal name(s) (RED_2) to the state(s) (S_0, S_1, S_2 and S_3) where it is asserted (see Figure 46). For simplicity, we are only considering the outputs which are asserted. Other outputs in their default states must be accounted for in the final state machine representation.

The signal GRN_1 is kept asserted for two extra clock cycles (Figure 46) to allow the traffic to pass when changing the traffic pattern from direction 2 to direction 1. The state diagram representation (Figure 48) requires assigning unconditional transition (control sequencing) from one state to another, for a number of states (S_1, S_2 and S_3), depending upon the required *time delay*. It also requires assigning the signal name (GRN_1) to all three states, which results in signal assertion for the extra two clock cycles.

Once the design has been converted into a state diagram (Figure 49) or flowchart (Figure 50) representing all of its timing and functional requirements, the next stage involves conversion to state machine syntax for its textual representation.

State Machine Syntax

The PALASM 2 programming software design file (also called the PAL device Design Specification or PDS) allows both conventional Boolean equations and state machine design constructs. It also allows both Mealy and Moore types of designs with extensive logic minimization capabilities and easy menu-driven operation. Other programming software packages which provide similar design capabilities are also available from various vendors.

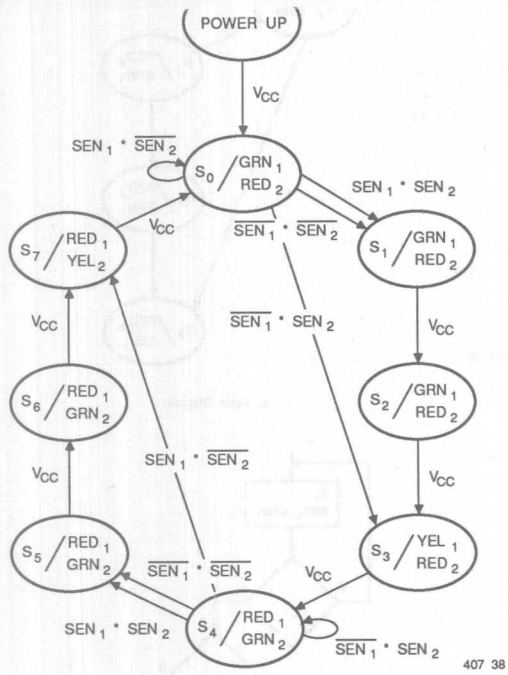


Figure 49. State Diagram—Traffic Signal Controller

A PALASM 2 state machine design file consists of four sections:

- Declaration section
- State section
- Condition section
- Simulation section (optional)

Declaration Section

As illustrated in Figure 51, this section stores the basic information about the device type and pin names. It is also used to store some bookkeeping information such as the company name and design revision numbers.

```
TITLE     TRAFFIC CONTROLLER
PATTERN   STATE MACHINE
REVISION  1
AUTHOR    J.  ENGINEER
COMPANY   MONOLITHIC MEMORIES
DATE      JANUARY 30, 1987

CHIP    S_MACHINE PMS14R21
        CLOCK DCLOCK SEN1 SEN2 I2 I3 I4
        I5 I6 I7 SDI GND
        RESET SDO RED1 YEL1 GRN1 RED2
        YEL2 GRN2 O1 O0 MODE VCC
```

Figure 51. Declaration Section

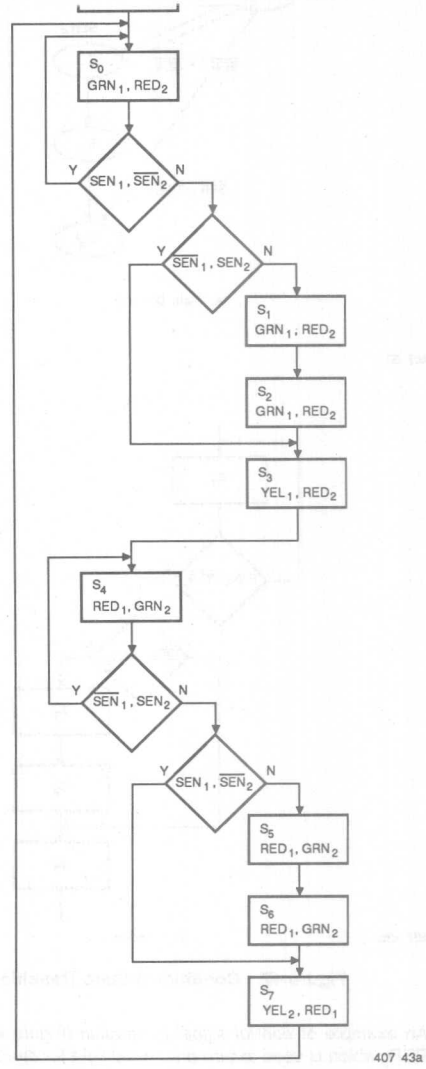


Figure 50. Flowchart—Traffic Signal Controller

State Section

The state section contains all the control sequencing and output generation information. The state section also stores all of the state initialization values and default parameters. As illustrated in Figure 52, it initially defines the type of state machine being designed with keywords MOORE_MACHINE or MEALY_MACHINE. For the PROSE device it also selects the use of pin 13 as either MASTER_RESET or OUTPUT_ENABLE.

It also defines the default output logic values as HIGH or LOW with the keyword DEFAULT_OUTPUT. Once an output is assigned a default logic value, it is explicitly mentioned only when asserted; otherwise it is assumed to be in its default logic value. This usually simplifies the task of drawing a state diagram (see Figure 49). Similarly, the keyword OUTPUT_HOLD defines the outputs which retain their previous values unless explicitly mentioned in a new state. The last two keywords also help isolate the user from the implementation details of J-K or D-type flip-flops when using PAL-device-based state machines. The keyword OUTPUT_HOLD is not used in this design.

```
STATE

  MOORE_MACHINE
  MASTER_RESET
  DEFAULT_OUTPUT /RED1/YEL1/GRN1/RED2/YEL2/GRN2
  POWER_UP := VCC -> S0
```

Figure 52. State Section

Once the default parameters are defined, all of the state transitions are represented. Only state names are used, since the state values are assigned automatically by the software. Every state transition (Figures 53 and 54) can depend upon the present state for direct control sequencing (state_x := VCC -> state_y), or the present state and input conditions for conditional control sequencing (state_x := Condition_name -> state_y). The syntax has a direct relationship to the state diagrams. For conditional control sequencing, the syntax also allows a default state when none of the conditions of transitions to other states are satisfied.

The output generation can also be represented easily. Outputs are specified from each present state. These outputs can be Moore type (state.OUTF := output_name) or Mealy type (state.OUTF := Condition_name -> output_name), depending upon whether or not conditions are used to generate outputs.

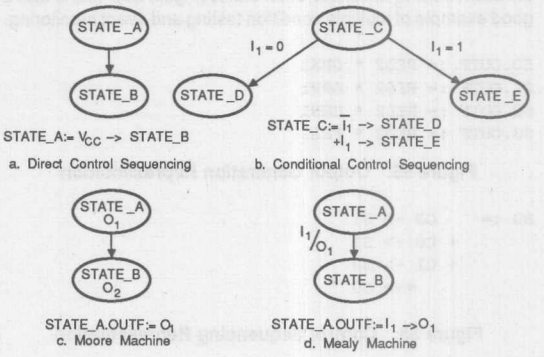


Figure 53. Direct Relationship Between State Diagram and State Syntax

Condition Section

This is the third section of the PALASM 2 state machine file structure. This section assigns names to all of the input conditions used by the state section for both control sequencing and output generation.

Design Example Syntax

All of the state machine tasks represented by the state diagram can be converted easily to textual format. The transition from state S_1 to S_2 and S_3 (Figure 55) illustrates how outputs RED_2 and GRN_1 are generated. These represent control signal generation and timing delay tasks of a state machine. Similarly, the arbitration function can be converted to textual format, showing the

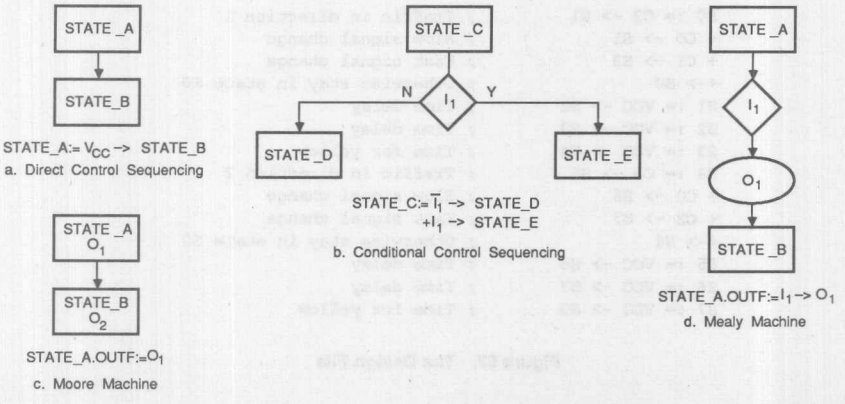


Figure 54. Direct Relationship Between Flowchart and State/Condition Syntax

transition from S_0 to various other states (Figure 56). This is also a good example of multiple condition testing and event monitoring.

```
S0.OUTF := RED2 * GRN1
S1.OUTF := RED2 * GRN1
S2.OUTF := RED2 * GRN1
S3.OUTF := RED2 * YEL1
```

Figure 55. Output Generation Representation

```
S0 :=    C3 -> S1
       + C0 -> S1
       + C1 -> S3
         +-> S0
```

Figure 56. Control Sequencing Representation

Simulation Section

The simulation section is similar to that used for conventional Boolean equation simulation. Please see page 4-137 for details.

Assembly and Programming

Once the state diagram is converted to its textual constructs, the file (Figure 57) can be easily assembled by using PALASM 2 software. The software then generates the device programming JEDEC format file. This JEDEC format file can be downloaded to a device hardware programmer provided by various vendors. For detail please see page 4-169 of the PALASM 2 software section.

Traffic Signal Controller

```
TITLE     TRAFFIC CONTROLLER
PATTERN   STATE MACHINE
REVISION  1
AUTHOR    J.  ENGINEER
COMPANY   MONOLITHIC MEMORIES
DATE      JANUARY 30, 1987

CHIP    S_MACHIN PMS14R21
        CLOCK DCLOCK SEN1 SEN2 I2 I3 I4
        I5 I6 I7 SDI GND
        RESET SDO RED1 YEL1 GRN1 RED2
        YEL2 GRN2 01 00 MODE VCC

STATE

  MOORE_MACHINE          ; Defined as a Moore Machine
  MASTER_RESET           ; Initialization
  DEFAULT_OUTPUT /RED1/YEL1/GRN1/RED2/YEL2/GRN2
  POWER_UP:=VCC -> S0    ; Power up state defined

S0 := C3 -> S1           ; Traffic in direction 1
+ C0 -> S1               ; Slow signal change
+ C1 -> S3               ; Fast signal change
+-> S0                   ; Otherwise stay in state S0
S1 := VCC -> S2          ; Time delay
S2 := VCC -> S3          ; Time delay
S3 := VCC -> S4          ; Time for yellow
S4 := C3 -> S5           ; Traffic in direction 2
+ C0 -> S5               ; Slow signal change
+ C2 -> S7               ; Fast signal change
+-> S4                   ; Otherwise stay in state S0
S5 := VCC -> S6          ; Time delay
S6 := VCC -> S7          ; Time delay
S7 := VCC -> S0          ; Time for yellow
```

Figure 57. The Design File

```
S0.OUTF := GRN1 * RED2        ; Allow direction 1 traffic
S1.OUTF := GRN1 * RED2
S2.OUTF := GRN1 * RED2
S3.OUTF := YEL1 * RED2        ; Change from direction 1 to 2
S4.OUTF := RED1 * GRN2        ; Allow direction 2 traffic
S5.OUTF := RED1 * GRN2
S6.OUTF := RED1 * GRN2
S7.OUTF := RED1 * YEL2        ; Change from direction 2 to 1

CONDITIONS

C0 = /SEN1 * /SEN2            ; Condition for no traffic
C1 = /SEN1 * SEN2             ; Condition for direction 2
C2 = SEN1 * /SEN2             ; Condition for direction 1
C3 = SEN1 * SEN2              ; Condition for traffic in both directions

SIMULATION

TRACE_ON CLOCK SEN1 SEN2 RED1 YEL1 GRN1 RED2 YEL2 GRN2

SETF RESET /CLOCK
CLOCKF CLOCK                  ;STATE TRANSITION ONLY ON 1ST CLOCK
CLOCKF CLOCK                  ;LIGHTS CHANGE ON 2ND CLOCK
CHECK /RED1 /YEL1 GRN1 /YEL2 /GRN2 RED2
SETF /SEN1 /SEN2
CLOCKF CLOCK
CLOCKF CLOCK
CHECK /RED1 /YEL1 GRN1 RED2 /YEL2 /GRN2
CLOCKF CLOCK
CHECK /RED1 YEL1 /GRN1 RED2 /YEL2 /GRN2
CLOCKF CLOCK
CHECK RED1 /YEL1 /GRN1 /RED2 /YEL2 GRN2
CLOCKF CLOCK
CHECK RED1 GRN2
CLOCKF CLOCK
CHECK RED1 YEL2
CLOCKF CLOCK
CHECK /RED1 /YEL1 GRN1 RED2 /YEL2 /GRN2
SETF /SEN1 SEN2
CLOCKF CLOCK
CHECK /RED1 /YEL1 GRN1 RED2 /YEL2 /GRN2
CLOCKF CLOCK
CLOCKF CLOCK
SETF SEN1 /SEN2
CLOCKF CLOCK
CLOCKF CLOCK
CHECK YEL2 RED1
CLOCKF CLOCK
CHECK GRN1 RED2

TRACE_OFF
```

Figure 57. The Design File (Cont'd.)

IMPORTANT
NOTES

Microprocessor-Based Systems

Introduction

The microprocessor has become the standard processing element in a wide variety of digital systems. Applications ranging from ignition control in automobiles to high performance engineering workstations utilize microprocessors. Usually a microprocessor with its associated VLSI components (the dynamic RAM, EPROM and standard microprocessor peripherals) can implement most application problems. The microprocessor's general-purpose nature, ease of use, and cost-effectiveness makes it the device of choice in all but the most performance-intensive or specialized applications.

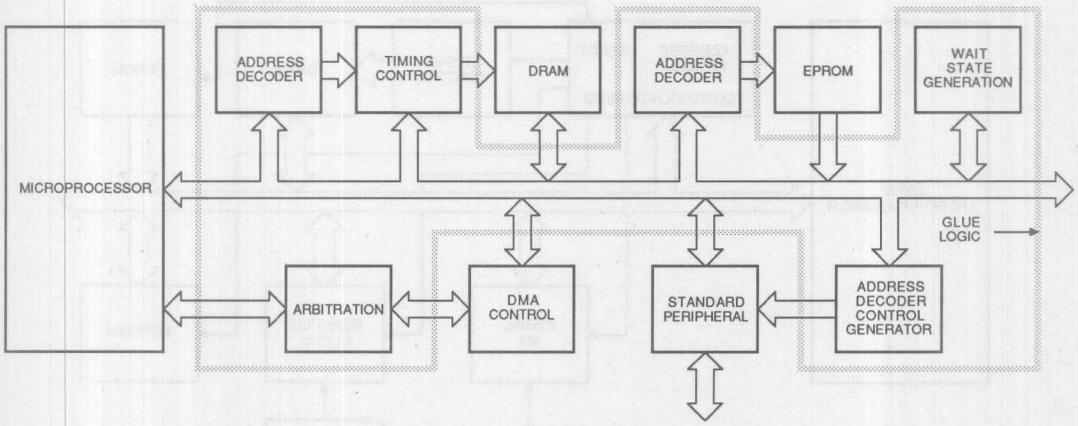
All the designer needs is the necessary "glue logic" to connect these VLSI components together. A typical microprocessor based design is illustrated in Figure 1. The heart of the design is the microprocessor. Typically, DRAMs provide the data storage and EPROMs provide the non-volatile program storage. Depending on the application such standard microprocessor peripherals as DMA controllers, serial I/O or disk controllers are usually present. Glue logic is needed to connect these devices together to create a working system. For example, memories and peripherals need to be selected based on the desired address range, a function called "address decoding"; high-speed address strobe signals RAS and CAS must be generated for the DRAMs; wait states must be generated for the slower memory and peripheral chips; arbitration and prioritization between different bus masters (system components which control the bus transactions: DMA and CPU) is necessary in multi-master systems.

Programmable Logic Devices as Standard Product Glue

The typical microprocessor system shown in Figure 1 highlights several of the places where programmable logic is used to implement the "glue" in a microprocessor system. These simple random logic and control-oriented functions are necessary to hold the standard products together to construct a complete system; thus the name "glue". The common glue functions in a microprocessor system share several application needs, and are best satisfied by programmable logic devices.

Microprocessor glue functions compete for board space with LSI devices in almost all cases. If the glue logic uses too much board space, the design will need to sacrifice features (reducing the design's value to the customer) or add extra boards (increasing the cost). In either case the cost/value ratio increases, which is undesirable. Programmable logic devices can reduce the board space required for glue functions because of the density advantage the devices have over standard SSI/MSI devices. This makes more board space available for standard VLSI devices or customized peripherals.

Since the microprocessor glue functions tend to be control- and timing-oriented, they are subject to many changes as the board design evolves, processor speeds increase, or new standard products are incorporated in the design. In complex systems, logic functions are commonly "tuned" to optimize bus band-width, peripheral access time, or memory timing only after the system is



409 01

Figure 1. Typical Microprocessor System

running. The optimum balance is usually too difficult to predict and can only be found via experimentation. Because of their programmable nature, PLDs offer the designer the capability of changing the logic at any time. Thus designs can quickly adjust to changing requirements, or can be improved without adding components or changing the circuit board.

The address decoding, timing, and simple state machines are "shallow" and "wide" logic functions. Shallow logic functions need only a few levels of gating at a high speed. The standard AND-OR array architecture offers 3 to 5 levels of gating (inverter, AND-gate, OR-gate, flip/flop-master and flip/flop-slave) at very high speed (10 to 18ns). Wide logic functions require a large number of inputs on each gating structure. For example, address decode functions may need over 10 inputs to decode a microprocessor address and an 8-bit refresh counter would require over 8 inputs for the most significant bit. The wide-input structure of PLDs, where practically all signals are available (in both polarities) at each gating structure, is a good match for this application requirement.

PLDs offer the designer several important charcteristics which match well with the needs of glue logic functions. Some common application areas will help illustrate the advantages of PLDs in microprocessor glue applications.

GLUE LOGIC NEEDS:	PROGRAMMABLE LOGIC ATTRIBUTES:
• Minimum Board Space • Easy and Frequent Design Changes • High Speed Simple Logic Functions	• High Logic Density • Quick and Easy Device Modifications • Optimized for Fast Shallow Logic

Figure 2. Table for Glue Logic Needs and Programmable Logic Attributes

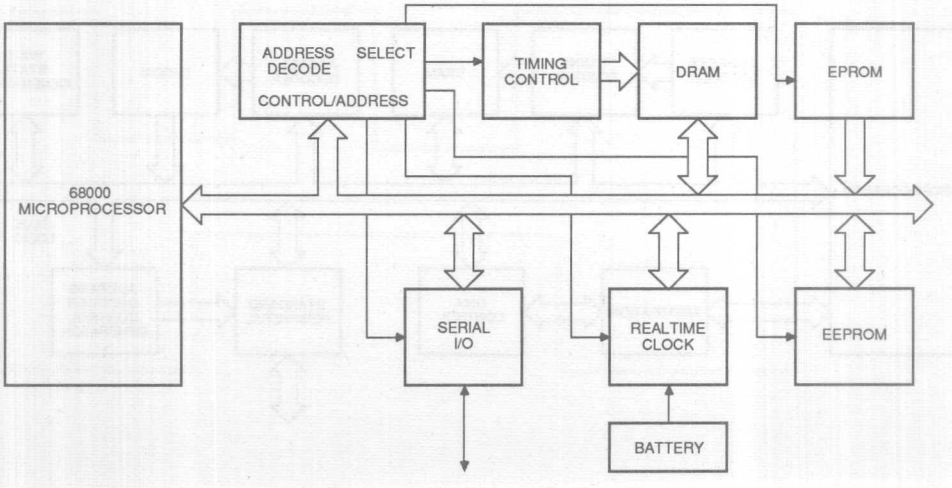
Address Decoding

The single most common application of combinatorial PAL devices is address decoding (Figure 3). In order for a microprocessor to communicate with a peripheral or memory chip a chip select signal must be generated by the glue logic. Usually a combination of address signals and control signals are required to generate the chip select for microprocessor memory and peripheral devices. For example, in the 68000 microprocessor some of the address signals, the address strobe signal, and the read/write signal must participate in the address decode function. The logic needed to generate the chip select is many times simply a NAND of the true or complement of the necessary address and control inputs.

In high-performance microprocessors, the chip select must be generated fast enough to allow single cycle access (no wait states) to memory. PLDs provide the high-speed wide-gating logic functions necessary to generate these chip select signals. Additionally, the addresses for peripherals and memory devices are often not known at the beginning of the design or may change many times during the design. PLDs can be easily changed to adjust to such modifications. More on address decoding is on page 2-35.

Wait-State Generation

Wait-state generation is in many ways related to the address decode function (Figure 4). After the addressed chip is selected, the CPU must be notified that the data transfer can be completed. The signal to the CPU which identifies the end of a data transfer may need to be extended if a slow device is being accessed. DRAMs, EPROMs, EEPROMs, and microprocessor peripherals may all need different delays depending on their access time bus bandwidth requirements and frequency of access. Usually there is at least a fast access (no wait-states) a slower access (one wait-state) and a longer access (several wait-states). The wait state



409 02

Figure 3. Address Decoding Application Example

generation logic must create a variable delay depending on the device being presently accessed. Usually the delay only depends on the present access and is independent of previous accesses.

A registered PAL device has the capability of implementing the simple state machine needed for wait-state generation. Additionally, the PLD contains enough logic to provide for a variety of memory, peripheral and processor speeds, without requiring additional board space. Thus a single design can have an extended lifetime, or provide a quick time to market when faster or lower-cost processors, peripherals and memories are available.

DRAM Control

DRAM control logic consists of several distinct functions: address multiplexing, RAS/MS/CAS generation, refresh request generation, and memory request arbitration. Address multiplexing, as the name implies, involves multiplexing the upper and lower processor addresses to the DRAM chips. RAS/MS/CAS generation involves the generation of the three separate DRAM timing signals: Row Address Strobe, Column Address Strobe, and Mux Select (on the address multiplexer). Tight timing relationships between the address multiplexer and the RAS/MS/CAS logic is necessary in high-performance systems. Refresh request generation logic is used to time the interval between DRAM refresh cycles. The logic is usually a wide counter using a slow clock. Memory request arbitration logic decides whether the refresh request will be granted or a memory access will be allowed.

PLDs have a variety of characteristics which make them well suited for DRAM control applications. High speed and minimum skew specifications are important considerations for address multiplexing, RAS/MS/CAS generation and refresh arbitration in high performance designs. High-speed PLDs have significantly tighter skew specifications than the equivalent multi-device SSI/MSI solutions. This makes PLD-based DRAM control applications easier to design and higher performance than SSI/MSI based designs. Detail on DRAM designs is provided on page 2-179.

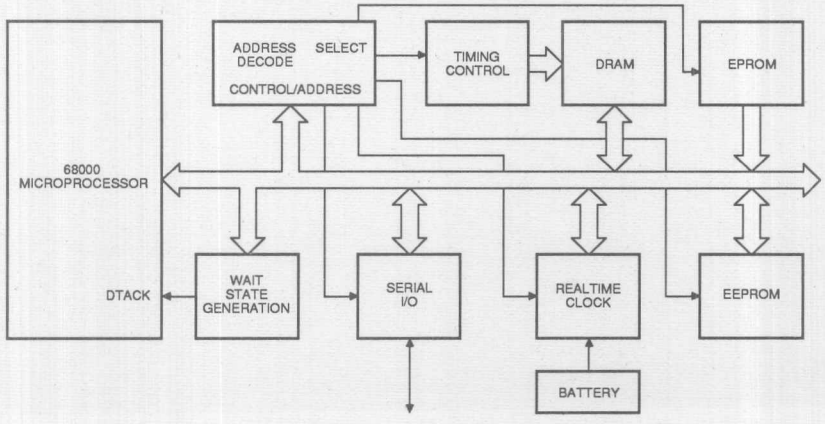
Microprocessor Peripheral Interface

Microprocessors are usually connected to a number of VLSI peripherals performing various ancilliary functions. In many cases a design calls for the use of "incompatible" microprocessor and VLSI peripherals. Cost, performance, availability, backwards compatibility, or other constraints sometimes dictate that different families of devices need to coexist in the same design. Glue logic is necessary to allow these incompatible devices to communicate. Sometimes logic signals need to be translated. The most obvious case is probably the different way a processor read or write cycle is encoded. One popular processor uses two separate read and write signals (/R or /W) with the active state indicating the presence of a valid address. Another popular processor uses an "address valid" signal to indicate the presence of an address and a single read/write signal to indicate the type of access. Peripherals hooking onto the other processor's bus will need some simple logic to translate the meaning of these signals. Often signal timing is also different between families. This requires signals to be delayed or qualified by glue logic.

The two most popular processor families for high performance designs are the 8086/80186/80286 family and the 60000/68010/68020 family. Each processor has its own peculiarities which the designer must take into account when hooking up a peripheral device. Subsequent discussion covers several applications examples for each family. Finally, PLDs can also be used to make custom microprocessor peripherals. An application example of a PLD-based interrupt controller is shown on page 2-172.

Conclusion

PLDs offer the designer several important characteristics which match well with the needs of glue logic applications. The following application examples will illustrate the PLD's versatility at implementing microprocessor glue functions.



409 03

Figure 4. Wait State Generator Application Example

Interfacing to the 8086/80186/80286

Overview

The 8086 is a general-purpose 16-bit microprocessor CPU. The CPU has a 16-bit data bus multiplexed with 16 address outputs. There are 4 additional address lines (segment addresses which are multiplexed with STATUS) that increase the memory range to 1 Mbyte. 8086 addresses are specified as bytes. In a 16 bit word, the least significant byte has the lower address and the most significant byte has the higher address. This is compatible with 8080, 8085, Z80 and PDP11 addressing schemes but differs from the Z8000 and 68000 addressing.

The data bus is "asynchronous", i.e., the CPU machine cycle can be stretched without clock manipulation by inserting Wait states between t2 and t3 of a read or write cycle to accommodate slower memory or peripherals. Unlike the 68000, the 8086 has separate address space for I/O (64 kBytes).

The 8086 can operate in MIN. or MAX. mode. Maximum mode offloads certain bus control functions to a peripheral device and allows the CPU to operate efficiently in a co-processor environment. A brief discussion on both the MIN. and MAX. modes are as follows:

MIN. mode: I/O addressing is defined by a HIGH or the IO/$\overline{M}$ output, and activated by the $\overline{RD}$ output for reading from memory or I/O, or activated by the $\overline{WR}$ output for writing to memory or I/O.

DMA: The Bus is requested by activating the HOLD input to the 8086. Bus Grant is confirmed by the HLDA output from the 8086.

MAX. mode: (8086 plus 8288) I/O operation is controlled by two outputs from the 8288.

- $\overline{IORC}$: active during Read from I/O
- $\overline{IOWC}$: active during Write to I/O
- $\overline{MRDC}$: active during Read from memory
- $\overline{MWTC}$: active during Write to memory

DMA: The Bus is requested and Bus Grant is acknowledged on the same pin ($\overline{RQ}$/GT0 OR $\overline{RQ}$/$\overline{GT1}$) through a pulsed handshake.

Interrupts in Min. and Max. Modes:

Interrupt is requested by activating the INTR or NMI inputs to the 8086.

Interrupt is acknowledged by the $\overline{INTA}$ pin on a MIN. mode 8086 or by the $\overline{INTA}$ pin on the 8288 in MAX. mode.

Note: There is no $\overline{RD}$ or $\overline{IORC}$ during the interrupt-acknowledge sequence.

8086 and Am7990 LANCE Interface

8086 and Am7990 LANCE

The LANCE, Am7990, has been designed to be interfaced easily with the popular 16-bit microprocessors (8086/80186, 68000, Z8000, LSI-11). Most of the interface logic is embedded inside the chip and is program selectable.

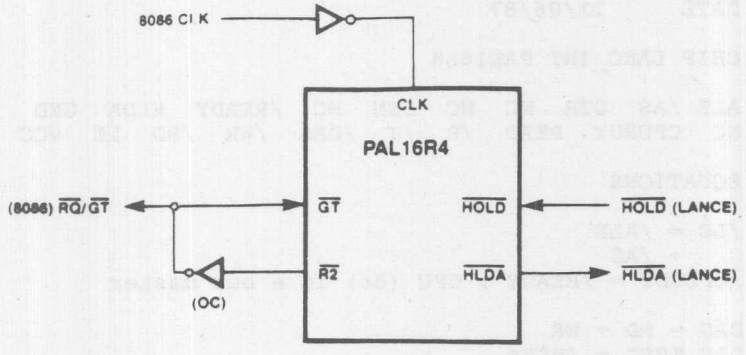
Although the LANCE itself has a multiplexed bus, it can easily be interfaced to demultiplexed buses with a minimal amount of effort. The following designs assume that the processor and the LANCE reside on the same board. Address buffers and data transceivers are set up to be shared between the processor and the LANCE. All of these designs use PAL devices to reduce the parts count.

The 8086 to LANCE interface requires a different Bus Request handshake, depending on whether the 8086 is configured in MAX. or MIN. mode. The 8086 has a bidirectional signal for both Bus Request and Bus Grant ($\overline{RQ}/\overline{GT}$). Both Bus Request ($\overline{RQ}$), and Bus Grant ($\overline{GT}$) to/from 8086 are one CPU clock wide, and are synchronous to the CPU clock. Figure 1 shows a PAL device design for the conversions in MAX. mode. This PAL device is utilized to include other external logic requirements for interfacing the LANCE to 8086. The interface diagram is similar to the one for the 80186 to LANCE interface except for the changes made in programming the PAL device. The interface timing diagram is shown in Figure 3. The PAL device design file is shown in Figure 2.

Figure 4 shows a block diagram of the 8086 to Am7990 interface, in MIN. mode. The interface also employs a PAL device to minimize parts count. The PAL device equations are given in Figure 5.

2

Figure 1. 8086 $\overline{RG}/\overline{GT}$, Am7990 $\overline{HOLD}/\overline{HOLDA}$ Conversion PAL Device

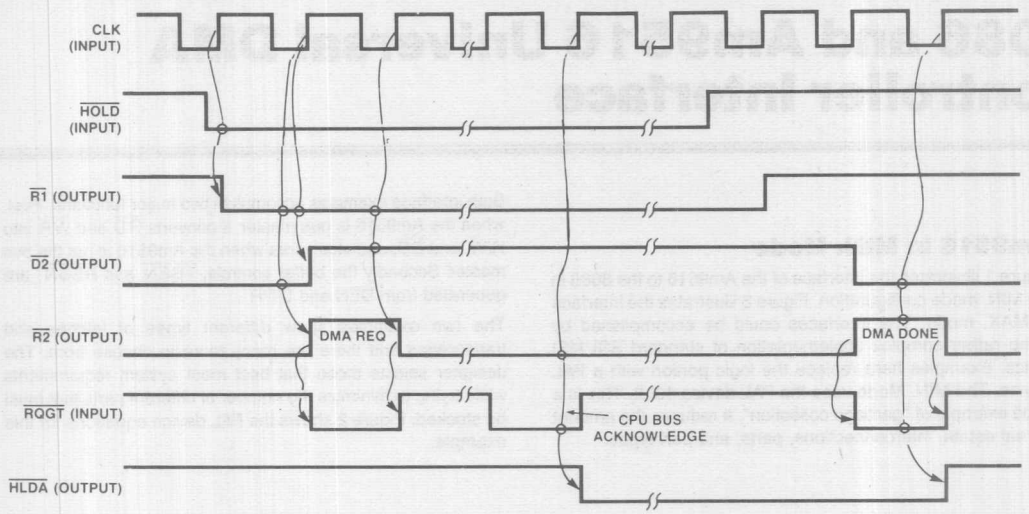


```
TITLE     8086 RQ/GT TO LANCE HOLD/HLDA CONVERTOR
PATTERN   PAT001
REVISION  01
AUTHOR    RASOUL OSKOUY
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP CONVERT PAL16R4

CLK NC HOLD /GT     NC  NC  NC NC NC GND
NC  NC NC   /HLDA /D2 /R2 /R1 NC NC VCC

EQUATIONS

R1 := HOLD    ;8086 RQ/GT conversion to LANCE HOLD/HLDA
D2 := /R1     ;both REQ and GRANT are one clock wide and
              ;are synchronous to CPU clock
R2 := R1*/D2
    + /R1*D2
HLDA := GT*/R2
      + HLDA*/D2
```

Figure 2. 8086 $\overline{RG}/\overline{GT}$, Am7990 $\overline{HOLD}/\overline{HLDA}$ Conversion PAL Device Design File

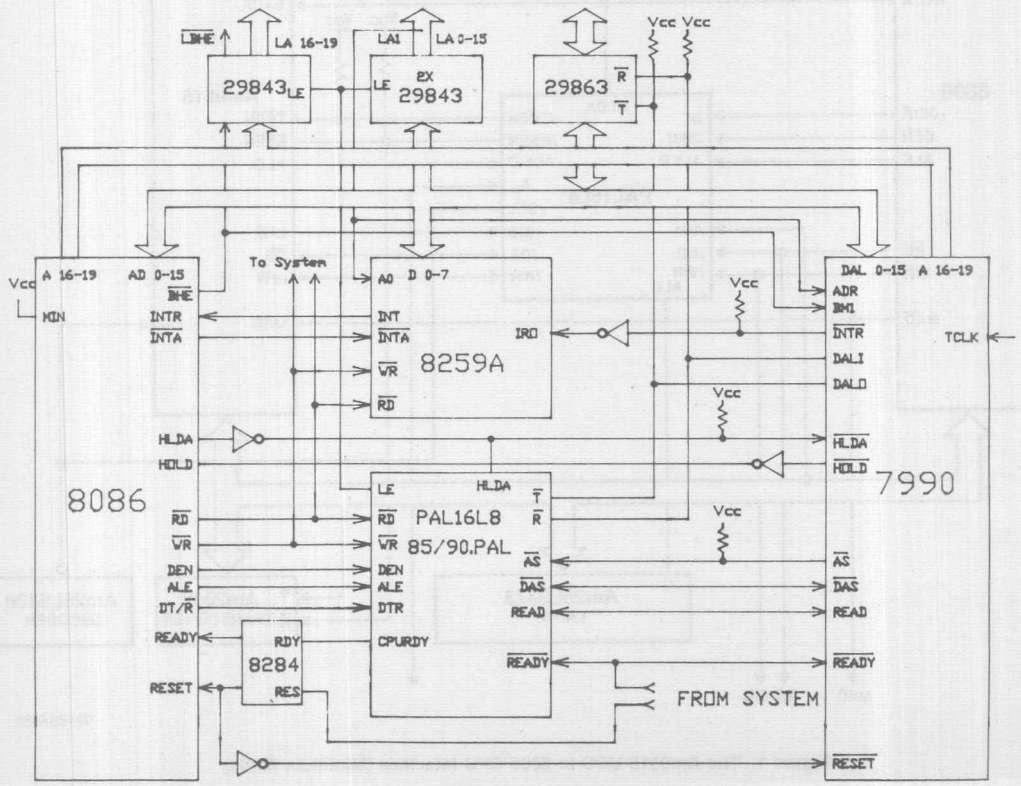
```
TITLE     8086 MIN MODE TO LANCE INTERFACE
PATTERN   PAT
REVISION  01
AUTHOR    KHUYNH NGUYEN
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP LANC_INT PAL16L8

ALE /AS  DTR  NC  NC  DEN  NC  /READY  HLDA  GND
NC  CPURDY  READ  /R  /T  /DAS  /WR  /RD  LE  VCC

EQUATIONS

/LE = /ALE
    + /AS
/CPURDY = /READY ; CPU (86) is a bus master

DAS = RD + WR
DAS.TRST = /HLDA
/READ = DTR
/READ.TRST = /HLDA
T = DTR
T.TRST = /HLDA
R = /DTR * DEN
R.TRST = /HLDA    ; LANCE is a bus master

RD = READ * DAS
RD.TRST = HLDA
WR = /READ * DAS
WR.TRST = HLDA
```

Figure 5. Source Listing for Example of Figure 4



02188A-20

Figure 3. AmPAL16R4, 8086 (Max. Mode) LANCE Interface Timing Diagram



02188A-21

Figure 4. Am7990 to 8086 Interface

8086 and Am9516 Universal DMA Controller Interface

Am9516 in MIN. Mode

Figure 1 illustrates the interface of the Am9516 to the 8086 in the MIN. mode configuration. Figure 3 illustrates the interface in MAX. mode. The interfaces could be accomplished by using rather complex implementation of standard SSI/MSI logics. Examples here replace the logic portion with a PAL device. The MIN. Mode uses the PAL device 16L8. This is a good example of "garbage collection". It reduces the amount of real estate, interconnections, parts, and part types.

Both interface examples accomplish two major functions. First, when the Am9516 is bus master, it converts $\overline{RD}$ and $\overline{WR}$ into R/$\overline{W}$ and $\overline{DS}$, and vice versa when the Am9516 is not the bus master. Secondly the buffer controls, $\overline{TBEN}$ and $\overline{REBN}$, are generated from DEN and DT/$\overline{R}$.

The two examples show different types of latches and transceivers and there are many more to choose from. The designer selects those that best meet system requirements while trying to minimize the number of different parts that must be stocked. Figure 2 shows the PAL device equations for this example.

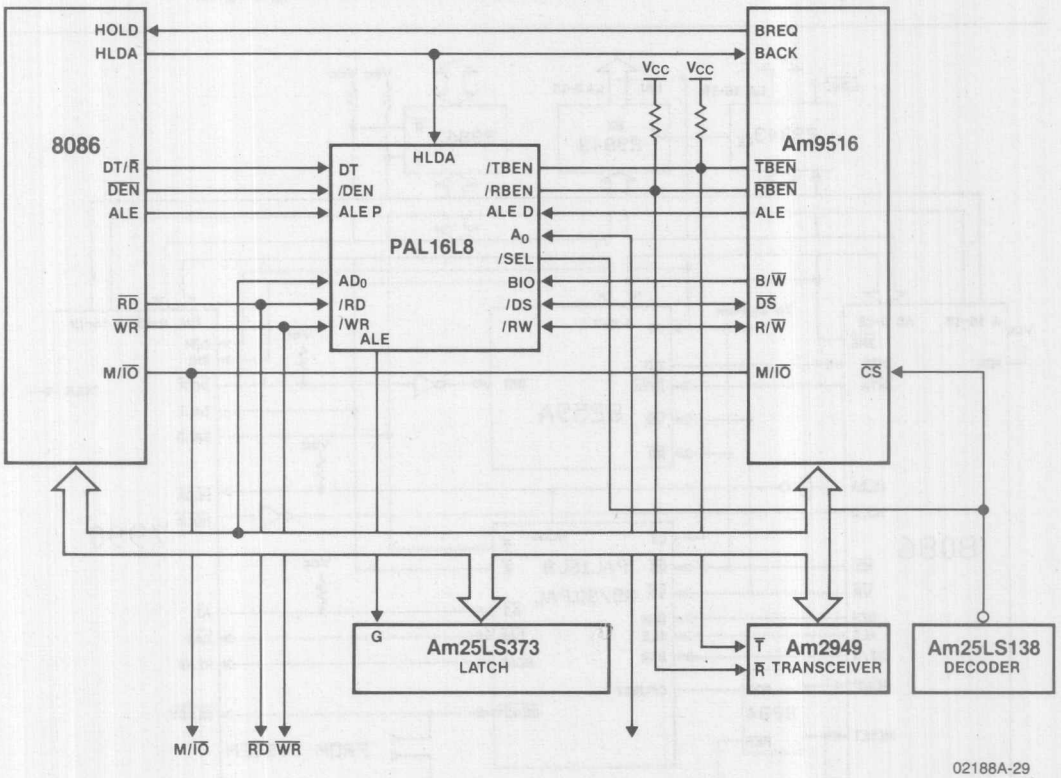


Figure 1. The Am9516 UDC to 8086 CPU Interface (Minimum Mode)

```
TITLE     Am9516 TO 8086 MIN MODE INTERFACE
PATTERN   01-3.66
REVISION  01
AUTHOR    JOE ENGINEER
COMPANY   ADVANCED MICRO DEVICES
DATE      11/05/87

CHIP DMA_INTFCE PAL16L8

;Am9516 to 8086 min mode interface chip

NC ALED ALEP HLDA BW AD0 DT /DEN /SEL GND
NC /RBEN /RD ALE A0 /RW /DS /WR /TBEN VCC

EQUATIONS

DS = RD + WR
DS.TRST = /HLDA

RW = DT
RW.TRST = /HLDA

TBEN = /DT * /SEL * DEN
TBEN.TRST = /HLDA

RBEN = DT * /SEL * DEN
RBEN.TRST = /HLDA

RD = /RW * DS
RD.TRST = /HLDA

WR = RW * DS
WR.TRST = /HLDA

ALE = /ALEP * /ALED

A0 =  /AD0 * /BW * HLDA * ALED
    + AD0 * BW * HLDA * ALED
    + /AD0 * /HLDA * ALEP
    + A0 * /ALEP + A0 * /ALED

;DESCRIPTION

;THIS PAL DEVICE CONVERTS THE CONTROL SIGNALS TO INTERFACE THE 8086
;IN MIN MODE TO THE Am9516 DMA CONTROLLER.  ANOTHER EXAMPLE SHOWS
;HOW THIS IS DONE IN MAX MODE.

; SIMULATION NOT INCLUDED
```

Figure 2. Source Listing for Example of Figure 1

Am9516 in MAX. Mode

This MAX. mode interface between the 8086 and Am9516 (Figure 3) also uses PAL devices to reduce component count.

The example makes several assumptions which result in a slightly more complex design than absolutely necessary. The 8086 is assumed to be in MAX. Mode, and the design has to be compatible with a MULTIBUS or similar interface; the 16R4 and 8288 could be eliminated if this is not the case.

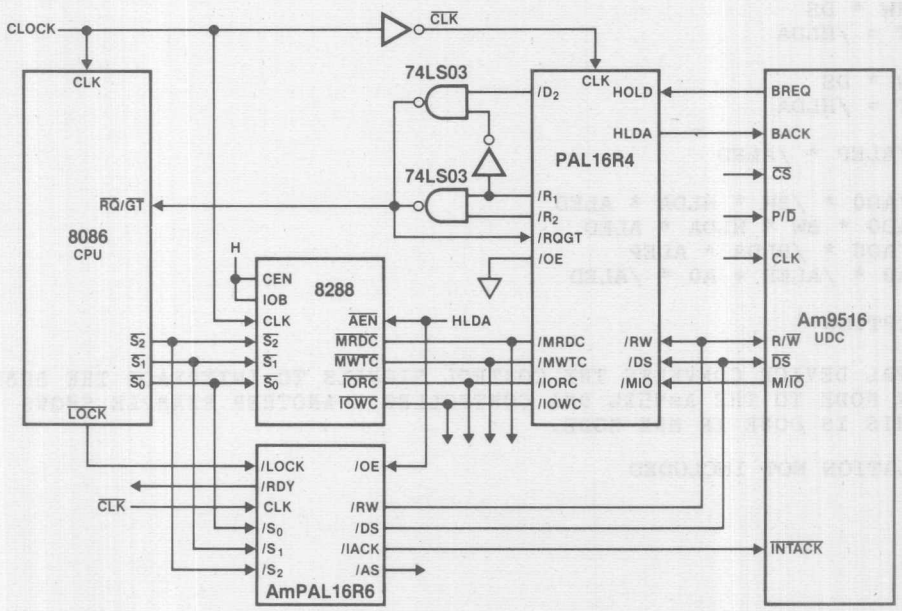
The 16R6 (Figure 4) is a PAL device that performs a function similar to the 8288, that is, it converts the processor's status signals and clock into control signals. In this case, the signals are R/$\overline{W}$, Data Strobe ($\overline{DS}$), Interrupt Acknowledge ($\overline{INTA}$) and Peripheral Acknowledge ($\overline{PACK}$). This PAL device basically generates ZBUS signals for Zilog peripherals. In the example, it connects to the Am9516 (UDC). The DMA Controller is very similar to the AmZ8016 except its bus interface has been modified to interface to non-multiplexed buses. This was changed from the previous design using a 16R8 to eliminate problems due to skew between OSC and CLK.

The 16R4, Figure 5, has two functions. The first is connecting the MIN. Mode protocol of the Am9516 or similar device to the MAX. Mode protocol for bus exchange. The 74LS03s are used to aid in this function. The timing waveform illustrates what happens in detail. The basic philosophy is that a rising edge on the HOLD input generates a pulse that is one clock wide. The CPU samples this pulse and, in response, issues a one-clock-wide pulse. The 16R4 uses this response pulse to generate HLDA. When the Am9516 has completed the necessary transfer, HOLD transitions HIGH to LOW. This generates another pulse to the CPU signalling that the Am9516 is done and that the CPU may continue.

The second function of the 16R4 is the conversion of R/$\overline{W}$, $\overline{DS}$ and M/$\overline{IO}$ into the MULTIBUS-compatible signals /MRDC, /MWTC, /IORC, and /IOWC, when HLDA is HIGH. It is possible to collapse the 74LS03s into the PAL device, thus reducing the external logic required to only one open collector inverter. The disadvantage, however, is that it adds two additional clocks for the bus exchange overhead, one during acquisition of the bus and one on bus release. For block transfers, this is not significant but it may be undesirable when performing single transfers, or short burst, or when in Demand Mode. To eliminate the 74LS03, change the equation for R2 to

/R1 • /D2 + R1 • D2,

and then drive the /[RQ/GT] line with a 74LS05 from the R2 output.



02188A-30

Figure 3.

```
TITLE    8086 TO 85XX PERIPHERAL INTERFACE
PATTERN  01-3.68
REVISION 01
AUTHOR   JOE BRCICH
COMPANY  ADVANCED MICRO DEVICES
DATE     11/06/87

CHIP DMA_INTRR PAL16R6

CLOCK  RESET  CLK   /S0   /S1  /S2  /LOCK   NC    NC   GND
/OE    /AS    /P1   /RW   /DS  /P0  /IACK  /RDY  CLKD  VCC

EQUATIONS

P0 := /RESET * S0 * /P0 * /P1
    + /RESET * S1 * /P0 * /P1
    + /RESET * S2 * /P0 * /P1
    + /RESET * S0 * P1
    + /RESET * S1 * P1
    + /RESET * S2 * P1

P1 := /RESET * /P0 * /P1
    + /RESET * P1 * S0
    + /RESET * P1 * S1
    + /RESET * P1 * S2

DS := /IACK * /P0 * P1 *  S0 * /S1 * S2
    + /IACK * /P0 * P1 * /S0 *  S1 * S2
    +  IACK *  S0 * S1 * S2 * P0 * /P1 * LOCK
    +  DS * S0  *  S1 * S2
    +  DS * S0  * /S1 * S2
    +  DS * /S0 *  S1 * S2

RW := S0 * /S1 * S2

IACK := /RESET * S0 * S1 * S2 * /P0 * /P1 * /LOCK
      + /RESET * IACK * S0 * S1 * S2 * P0 * /P1 * /LOCK
      + /RESET * IACK * LOCK * /DS
      + /RESET * IACK * /LOCK * DS * /P0 * P1

RDY := /RESET *  S0  * /S1 * S2 * P0 * P1
     + /RESET * /S0  *  S1 * S2 * P0 * P1
     + /RESET * RDY  *  S0 * /S1 * S2
     + /RESET * RDY  * /S0 * S1 * S2
     + /RESET * IACK *  S0 * S1 * S2 * DS
     + /RESET * RDY  *  S0 * S1 * S2

/CLKD = CLK

AS = /CLKD * P0 * /P1 * /IACK * CLK

;DESCRIPTION

;THIS PAL DEVICE TRANSLATES 8086 BUS SIGNALS INTO COMPATIBLE SIGNALS FOR
;THE 9516.  IT IS ALSO APPLICABLE TO 85XX PERIPHERALS BY ALTERING /RW AND
;/DS TO /RD AND /WR.  ONE FLIP-FLOP IS AVAILABLE TO GIVE THE NECESSARY
;DELAY TO THE FALLING EDGE OF /WR.  THE DATA STROBE TIMING FOR A WRITE
;CYCLE IS DELAYED UNTIL THE FALLING EDGE OF T2 TO MEET THE REQUIREMENTS
;OF THE 85XX PARTS.  THIS DESIGN ASSERTS RDY TO DEMAND ONE WAIT STATE
;FROM THE 8086.  THIS WAIT STATE IS NOT LONG ENOUGH FOR DESIGNS WHICH USE
;AN 8 MHz 8086.  THEREFORE, WITH AN 8 MHz CPU, 85XX PERIPHERALS SHOULD BE
;USED.  AS AN ALTERNATIVE, THREE WAIT STATES CAN BE USED BY ALTERING THE
;RDY EQUATION.  THIS PAL DEVICE ALSO TRANSFORMS THE 8086 TWO-CYCLE
;INTERRUPT ACKNOWLEDGE INTO A SINGLE CYCLE OF THE TYPE NECESSARY FOR 85XX
;PARTS.  THIS IS MADE POSSIBLE BY SAMPLING THE LOCK STATUS, P0, P1, AND
;IACK SIGNALS.
```

Figure 4. Source Listing for Example of Figure 3

```
TITLE     Am9516 TO 8086 INTERFACE
PATTERN   01-3.67
REVISION  01
AUTHOR    JOE ENGINEER
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP DMA_INTFCE2 PAL16R4

;8086 to Am9516 interface

CLK /RQGT HOLD   NC   NC  NC  /RW  /DS    MIO  GND
/OE /MWTC /MRDC HLDA /D2 /R2  /R1  /IOWC /IORC VCC

EQUATIONS

IORC = /MIO * DS * /RW
IORC.TRST = HLDA

IOWC = /MIO * DS * RW
IOWC.TRST = HLDA

MRDC = MIO * DS * /RW
MRDC.TRST = HLDA

MWTC = MIO * DS * RW
MWTC.TRST = HLDA

R1   := HOLD

R2   := /R1

D2   := R1

/HLDA := /R1 + /D2 * /HLDA +
         /RQGT * /HLDA

;DESCRIPTION

;THIS DEVICE CONVERTS THE MIN MODE SIGNALS HOLD AND HLDA TO THE
;MAX MODE RQGT PROTOCOL.  ADDITIONALLY IT GENERATES THE 8288
;EQUIVALENT CONTROL OUTPUTS MRDC, MWTC, IORC, AND IOWC.  THIS
;PAL DEVICE WAS USED TO CONNECT THE Am9516 TO THE 8086 IN MAX MODE.

; SIMULATION NOT INCLUDED
```

Figure 5. Source Listing for Example of Figure 3

80286 to Am9568 Data Ciphering Processor Interface

80286 to Am9568 Data Ciphering Processor Interface

This interface is designed for an 8-MHz CPU where the Data Ciphering Processor (DCP) is synchronously operating at the maximum clock rate of 4 MHz. A block diagram for the interface is shown in Figure 1. The Am9568 requires a narrower width of address strobe than the Am9518. This works comfortably with the 60-ns address strobe width of an 8-MHz CPU.

The MULTIBUS Mode Select input of the Bus Controller 82288 is tied LOW to optimize the command and control signals for short bus cycles. The Command Delay (CMDLY) becomes active-HIGH for one 16-MHz clock cycle whenever the DCP is selected to delay the Read and Write strobes by 125 ns. This satisfies the timing requirement of the minimum delay between ALE inactive and Read or Write strobe active of the DCP. An open-collector gate must be added to allow other peripherals to drive this input.

The ALE, $\overline{IORC}$ and $\overline{IOWC}$ outputs of the 82288 are wired directly to the DCP. ALE strobes a D-Flip-Flop to store the state of Chip Select for the entire cycle.

$\overline{Q}_3$ and the latched Chip Select CSL are ANDed externally to generate the Synchronous Ready for the 82284. The 82284 samples the line at the falling edge of the clock. The registered output $\overline{Q}_3$ is clocked with the rising edge of the same clock, thus satisfying the setup and hold time requirements of the 82284. Two Wait States are inserted.

Half of the PAL device operates as a bidirectional Address/Data Multiplexer. During the Address Latch Enable active phase, the state of A_1 and A_2 is transferred to the AD_1 and AD_2 pins of the PAL device. The DCP latches this two bit-address with the falling edge of ALE.

When $\overline{IORC}$ and CSL are active, the states of AD_1 and AD_2 are passed to D_1 and D_2 respectively. The DCP Register can be read. If $\overline{IOWC}$ and CSL are active, the data path is turned around: D_1 and D_2 are inputs, AD_1 and AD_2 are outputs.

The Address Hold Time of the PAL device is sufficient because the address information is passed to AD_1 and AD_2 whenever IORC*CSL or IOWC*CSL are not true, i.e. whenever data is not transferred between the CPU and the DCP.

The Read Data Hold Time requirement of 5 ns of the Am9568 is satisfied by the propagation delay of the PAL device.

The Read Data Hold Time requirement of 5 ns of the iAPX286 is also satisfied by the PAL device.

The Master Port Chip Select ($\overline{MCS}$) input of the DCP is connected to the unlatched address decoder output.

Figure 3 shows the source listing and the pin descriptions.

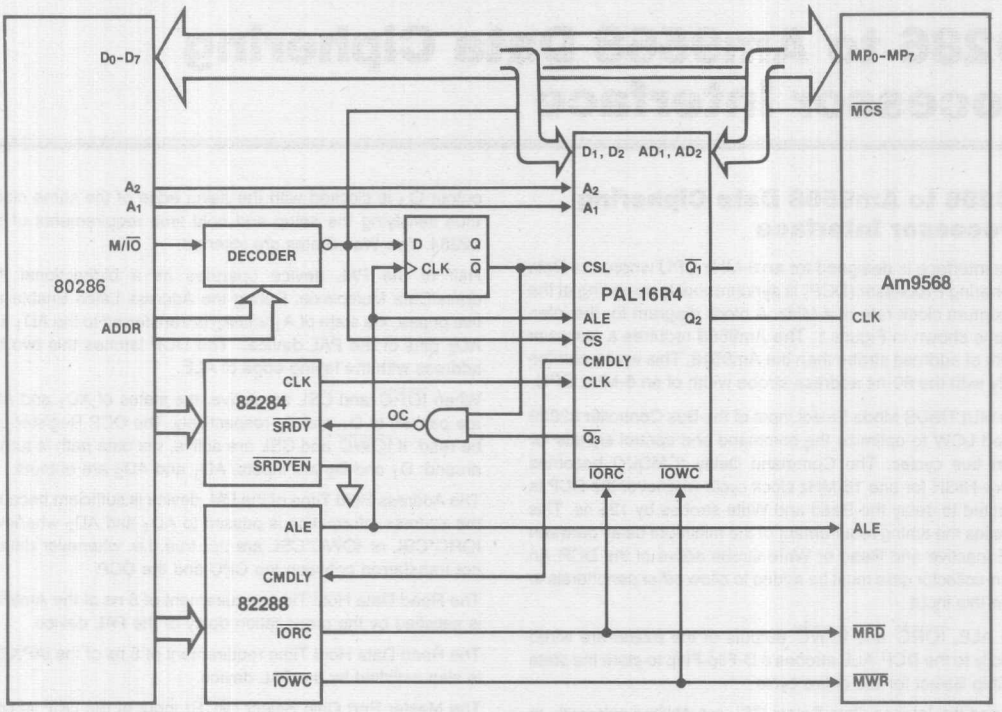


Figure 1. 80286 to Am9568 Interface

The DCP Clock

The PAL device synchronizes the DCP clock to the Data Strobes $\overline{IORC}$ and $\overline{IOWC}$ (Figure 2). It also divides the 16 MHz system clock (8 MHz CPU clock) down to the maximum DCP clock rate of 4 MHz. At this clock rate, the Data Strobe Delay to the DCP clock must be 0–30 ns. The Bus Controller is specified to generate a Data Strobe timing of 3–15 ns to the falling edge of CLK (16 MHz). Because of the higher propagation delay of a standard PAL device, the registered outputs are toggled at the rising edge of CLK before the Data Strobes become inactive. This gives an additional 32.5 ns for the DCP clock signal path.

Q_1 to Q_3 are three outputs of the PAL device state machine. The registered outputs are clocked with the rising edge of the 16-MHz 82284 clock. Whenever ALE and $\overline{CS}$ are active, Q_1 to Q_3 are set to the initial state. Q_1 to Q_3 are outputs of a 3-bit down counter, with Q_3 as the most significant bit.

Q_3 is used to generate the $\overline{SRDY}$ signal for the 82284 as mentioned above.

Q_2 is the DCP clock. This design must guarantee that the minimum DCP clock HIGH or LOW time is at least 115 ns or two 16-MHz clock cycles. This is done by toggling Q_2 only during phase 2 cycles of the CPU. The CPU design guarantees that there is always a phase 1 cycle between two phase 2 cycles.

Assuming a typical PAL device propagation delay of 25 ns, timing parameter tCDS (Time Clock Data Strobe) is 10.5 to 22.5 ns (3 + 32.5 − 25 ns to 15 + 32.5 − 25 ns). This satisfies the 0 to 30 ns requirement.

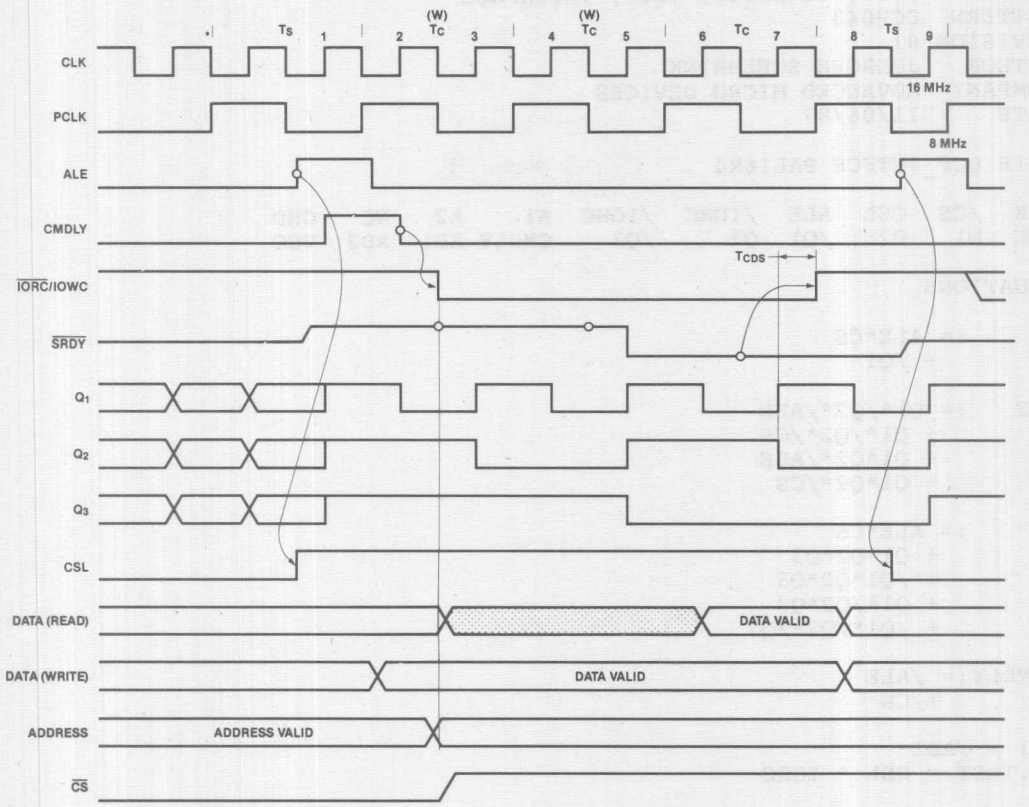
The AmPAL16R4 has active-LOW outputs. But one output, Q_2, should be active-HIGH. The equation for Q_2 was derived to be

$$Q_2 = ALE * CS + Q_1 * Q_2 + \overline{Q}_1 * \overline{Q}_2$$

To compensate for the inversion in the PAL device, the PALASM software minimizer (or 16RP4) can be used to convert it to the form shown in the PAL device Design Specification.

Improvements

The DCP needs two Wait States only when the Control Registers are read. Data Register read or writes and Control Register reads can be executed with only one Wait State, which improves the Data Ciphering speed of this interface. The more sophisticated Wait control logic and the two external TTL gates can be integrated into one PAL22V10 device.



02188A-44

Figure 2. Timing Diagram

```
TITLE     80286 TO Am9568 (DCP) INTERFACE
PATTERN   DCP043
REVISION  01
AUTHOR    JUERGEN STELBRINK
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP DCP_INTFCE PAL16R4

CLK   /CS   CSL   ALE   /IORC   /IOWC   A1     A2    NC    GND
/OE   D1    D2    /Q1   Q2      /Q3     CMDLY  AD1   AD2   VCC

EQUATIONS

Q1     := ALE*CS
        + /Q1

/Q2    := Q1*/Q2*/ALE
        + Q1*/Q2*/CS
        + Q1*Q2*/ALE
        + Q1*Q2*/CS

Q3     := ALE*CS
        + Q1*Q2*Q3
        + /Q1*Q2*Q3
        + Q1*/Q2*Q3
        + /Q1*/Q2*/Q3

/CMDLY:= /ALE
        +/CS

/D1 = /AD1
D1.TRST = CSL * IORC

/D2 = /AD2
D2.TRST = CSL * IORC

/AD1 = /A1*ALE + /D1*/ALE
AD1.TRST = CSL * IORC

/AD2 = /A2*ALE + /D2*/ALE
AD2.TRST = CSL * IORC

; SIMULATION NOT INCLUDED
```

Figure 3. Design File for the Example of Figure 1

80286 to Am8530 Interface

80286 to Am8530 Interface

The Am8530 is a high-speed, dual-channel serial communications controller that supports a variety of advanced communication protocols. It supports data rates up to 1.5M bps.

This design note shows an 80286 to an Am8530 CPU interface when no interrupts are used (Figures 1 and 2). The design is for a CPU running at 5 MHz with a 100 ns system clock cycle time. This clock is used to generate the clock for the 6 MHz Am8530 SCC.

The chip select generated by the external decoder in an 80286 system that is enabled by the M/IO line will go inactive with the falling edge of T_{CLK}-phase 1. However, the chip enable for the SCC should stay valid for the entire read or write cycle. The PAL device generates the chip enable signal for the SCC using a structure equivalent to two cross-coupled NOR gates.

The Am8530 requires a minimum pulse width of 250 ns for $\overline{RD}$ and $\overline{WR}$ control signals. For a 100-ns cycle time 80286 with no wait state, these controls will be active for 200 ns only. SRDY control is generated for the 82284 bus controller to guarantee minimum $\overline{RD}$ and $\overline{WR}$ pulse widths for the Am8530.

CMDLY is not used in this design because the address to read or write control setup time required is only 80 ns for the Am8530; with a 100 ns system cycle time, the time between ALE valid and /IORC or /IOWC valid is 100 ns. Even after accounting for the address latch delay, 80 ns timing will still be met.

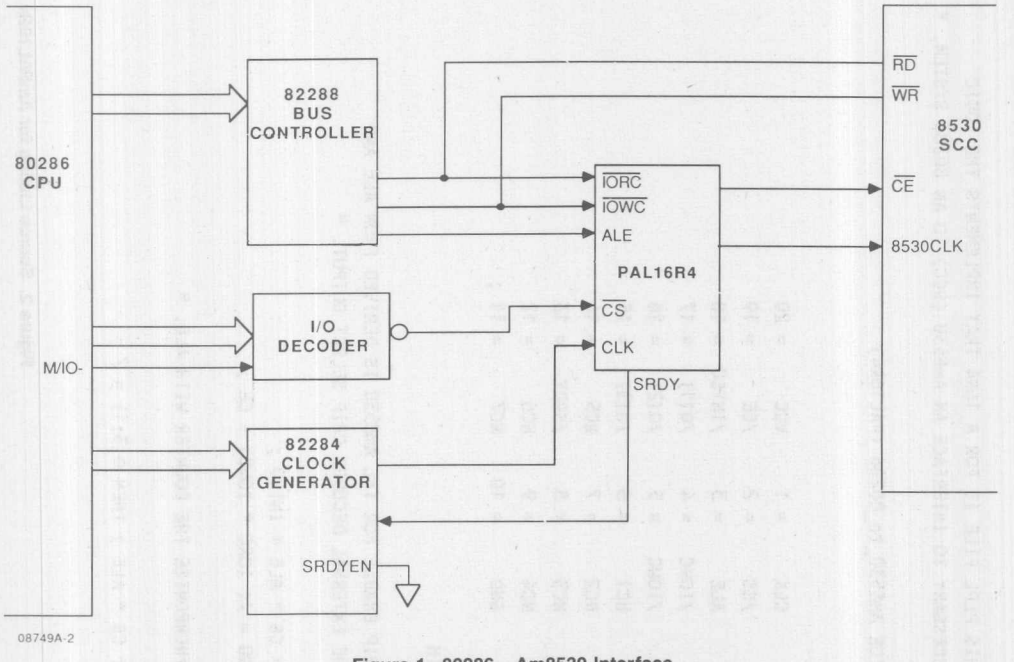


Figure 1. 80286—Am8530 Interface

```
" THIS PLPL FILE IS FOR A 16R4 THAT IMPLEMENTS THE LOGIC
  NECESSARY TO INTERFACE AN Am8530 (SCC) TO AN 80286 SYSTEM. "

DEVICE Am8530_to_80286 (PAL16R4)

PIN
        CLK     = 1     VCC     = 20
        /CS     = 2     /CE     = 19
        ALE     = 3     /INTSO  = 18
        /IORC   = 4     /Q[1]   = 17
        /IOWC   = 5     /Q[2]   = 16
        NC1     = 6     /Q[3]   = 15
        NC2     = 7     NC5     = 14
        NC3     = 8     /SRDY   = 13
        NC4     = 9     NC6     = 12
        GND     = 10    NC7     = 11 ;

BEGIN

" CHIP ENABLE FOR THE Am8530 IS DERIVED FROM ALE AND
  THE EXTERNAL DECODER CHIP SELECT OUTPUT. "

CE = CS * ALE + INTSO ;
INTSO = /( IORC * IOWC) + CE ;

" SYNCHRONIZE THE COUNTER WITH ALE. "

IF ( CS * ALE ) THEN Q[3:1] = 7 ;

" COUNT DOWN SPECIFICATION FOR THE COUNTER. "
" Q2 OF THE COUNTER IS THE CLOCK INPUT OF THE Am8530. "

CASE ( Q[3:1] )
       BEGIN
       0 ) Q[3:1] := 7 ;
       1 ) Q[3:1] := 0 ;
       2 ) Q[3:1] := 1 ;
       3 ) Q[3:1] := 2 ;
       4 ) Q[3:1] := 3 ;
       5 ) Q[3:1] := 4 ;
       6 ) Q[3:1] := 5 ;
       7 ) Q[3:1] := 6 ;
       END ;

" SRDY IS GENERATED FOR WAIT STATE GENERATION, IT IS SENT
  TO THE 82284 CLOCK GENERATOR. "

SRDY = /(Q[3] * CE) ;

END.
```

Figure 2. Source Listing for AmPAL16R4 for the Example of Figure 1

Interfacing to the 68000/68020

The 68000 has an asynchronous, 16-bit, bidirectional data bus. Data types supported by the 68000 are: bit data, integer data of 8, 16, or 32 bits, 32-bit addresses and binary-coded decimal data. It can transfer and accept data in either words or bytes. The $\overline{\text{DTACK}}$ input indicates the completion of a data transfer. When the processor recognizes $\overline{\text{DTACK}}$ during a read cycle, the data is latched and the bus cycle terminates. When $\overline{\text{DTACK}}$ is recognized during a write cycle, the bus cycle also terminates. An active transition of $\overline{\text{DTACK}}$ indicates the termination of a data transfer on the bus. All control and data lines are sampled during the 68000's clock HIGH time. The clock is internally buffered, which results in some slight differences in the sampling and recognition of various signals. The 68000 mask sets prior to CC1 and allows $\overline{\text{DTACK}}$ to be recognized as early as S2, and all devices allow $\overline{\text{BERR}}$ or $\overline{\text{DTACK}}$ to be recognized in S4, S6, etc., which terminates the cycle. If the required setup time is met during S4, $\overline{\text{DTACK}}$ will be recognized during S5 and S6, and data will be captured during S6. $\overline{\text{DTACK}}$ signal is internally synchronized to allow for valid operation in an asynchronous system. If an asynchronous control signal does not meet the required setup time, it is possible that it may not be recognized during that cycle. Because of this, synchronous systems must not allow $\overline{\text{DTACK}}$ to precede data by more than 40 to 240 nanoseconds, depending on the speed of the particular processor. I/O is memory-mapped, i.e., there are no special I/O control signals, any peripheral is treated as a memory location.

DMA: This Bus is requested by activating the $\overline{\text{BR}}$ input of the 68000. Bus Arbitration is started by the $\overline{\text{BG}}$ output going active. The Bus is available when $\overline{\text{AS}}$ becomes inactive. The requesting device must acknowledge bus mastership by activating the $\overline{\text{BGACK}}$ input to the CPU.

The 23-bit address ($A_1 \ldots A_{23}$) is on a unidirectional, three-state bus, and can address 8 M words (16 M bytes) of memory or I/O. It provides the address for bus operation during all cycles, except the interrupt cycles. During interrupt cycles, address lines A1, A2 and A3 provide information about the level of interrupt being serviced. Instead of A_0 and BYTE/$\overline{\text{WORD}}$, there are two separate data strobe lines for the two bytes in a word. A note of caution here, the 68000 treats the MSB of the lower byte as an even byte, or word address. The same goes with processors such as the Z8000. Processors such as the 8086 treats the lower byte as the odd byte.

Interrupt is requested by activating any combination of the interrupt inputs to the 68000 (IPL0...2), indicating the encoded priority level of the interrupt requester (inputs at or below the current processor priority are ignored). The 68000 automatically saves the status register, switches to supervisor mode, fetches a vector number from the interrupting device, and displays the interrupt level on the address bus. For interfacing with old 68000 peripherals, the 68000 issues an Enable signal at one-tenth of the processor clock frequency. There are a number of AMD proprietary third generation peripherals that can be interfaced to the 68000 CPU, to improve system performance. This section deals mainly with the interfacing of the 68000 and some of the AMD proprietary peripherals.

The 68000 to Am8530 Interface with Interrupts

This example shows how a PAL device simplifies the task of interrupt generation compared to the MSI implementation. The block diagram for the interface via a PAL device is shown in Figure 1. The timing diagram (Figure 2) illustrates the Interrupt Acknowledge cycle. As in the other designs, $\overline{RD}$ is generated during Interrupt Acknowledge to place the vector on the bus.

The timing during register programming is not shown. The PAL device allows selection of one or two Wait States by making W_0 HIGH or LOW, respectively. The table below shows the appropriate number of Wait States as a function of CPU speed.

Part	CPU Speed	Wait States
85XX	4 MHz	1
85XX	6 MHz	2
85XXA	8 MHz	2
85XXA	10 MHz	2

PAL device equations are shown in Figure 3.

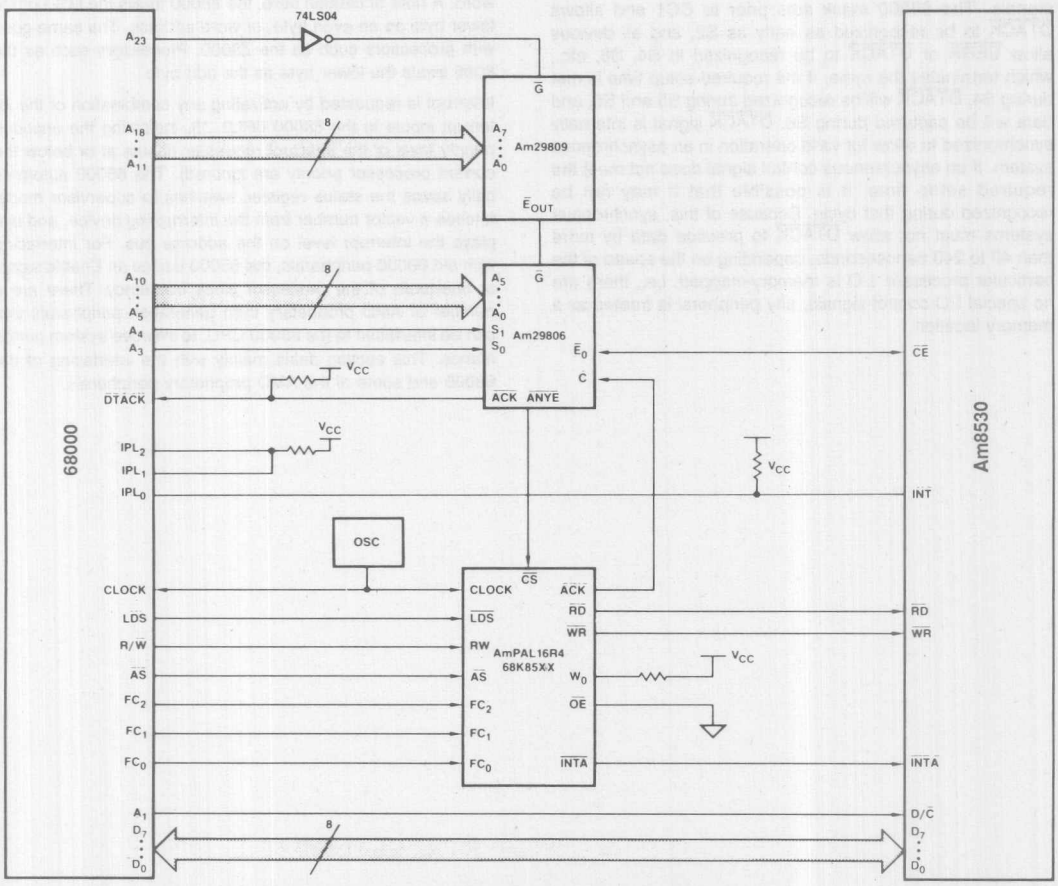


Figure 1. 68000 to Am8530 Connection Using a PAL Device

02188A-68

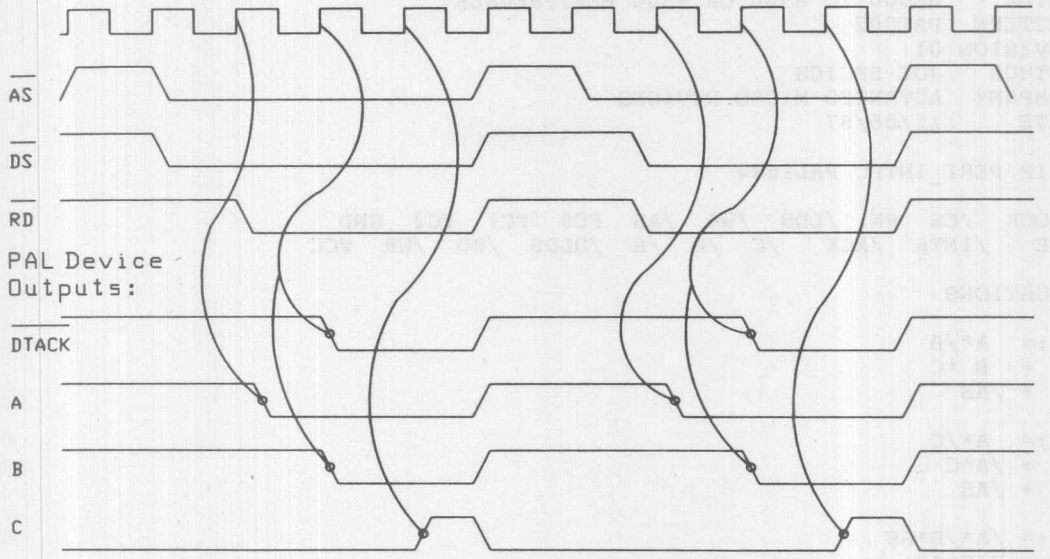


Figure 2. Timing Diagram for 68000 to Am8530 Interface

```
TITLE     68000 TO 8500 OR 9500 PERIPHERALS
PATTERN   PAT002
REVISION  01
AUTHOR    JOE BRCICH
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP PERI_INTFC PAL16R4

CLOCK  /CS  RW  /LDS  /W0  /AS  FC0  FC1  FC2  GND
/OE   /INTA  /ACK   /C  /B  /A  /DLDS  /RD  /WR  VCC

EQUATIONS

A :=  A*/B
   +  B *C
   + /AS

B :=  A*/C
   + /A*C
   + /AS

C := /A*/B*AS
   + B*C*AS

DLDS := LDS

RD = LDS*DLDS*RW*/INTA
   + A*C*INTA*AS
   + A*/B*INTA*AS

WR = LDS * /RW

INTA = FC0*FC1*FC2*AS

ACK = /INTA*/A*/B*/C*/W0
    + /INTA*/A*/B*C*W0
    +  INTA*/B*A
    +  ACK * LDS

;DESCRIPTION

;THIS PAL DEVICE INTERFACES 85XX TYPE PERIPHERALS TO THE 68000
;MICRO PROCESSOR.  IT INSERTS 1 OR 2 WAIT STATES AS SELECTED BY
;/W0=0 IS ONE AND /W0=1 IS TWO WAIT STATES.  FOUR WAIT STATES ARE
;INSERTED DURING INTERRUPT ACKNOWLEDGE CYCLES.  ALSO THE RD OUTPUT
;GENERATED DURING INTA IS A FUNCTION OF THE INTERNAL STATE
;MACHINE AND NOT A FUNCTION OF LDS.  OE CAN BE LEFT OPEN SINCE
;THE FLIP FLOP OUTPUTS ARE NOT USED DIRECTLY.  THE FALLING EDGE
;OF RD IS DELAYED IN ORDER TO GUARANTEE THE CS TO RD SETUP TIME
;REQUIREMENTS.

; SIMULATION NOT INCLUDED
```

Figure 3. Source Listing for the Example of Figure 1

68000 and Am7990 LANCE Interface

The design on the LANCE has made it easier for the user to interface the device with demultiplexed buses. The example shown here is an interface to be compatible with an 8 MHz or faster 68000 (Figure 1). The two flip-flops are needed to adapt the LANCE bus request handshake to the 68000.

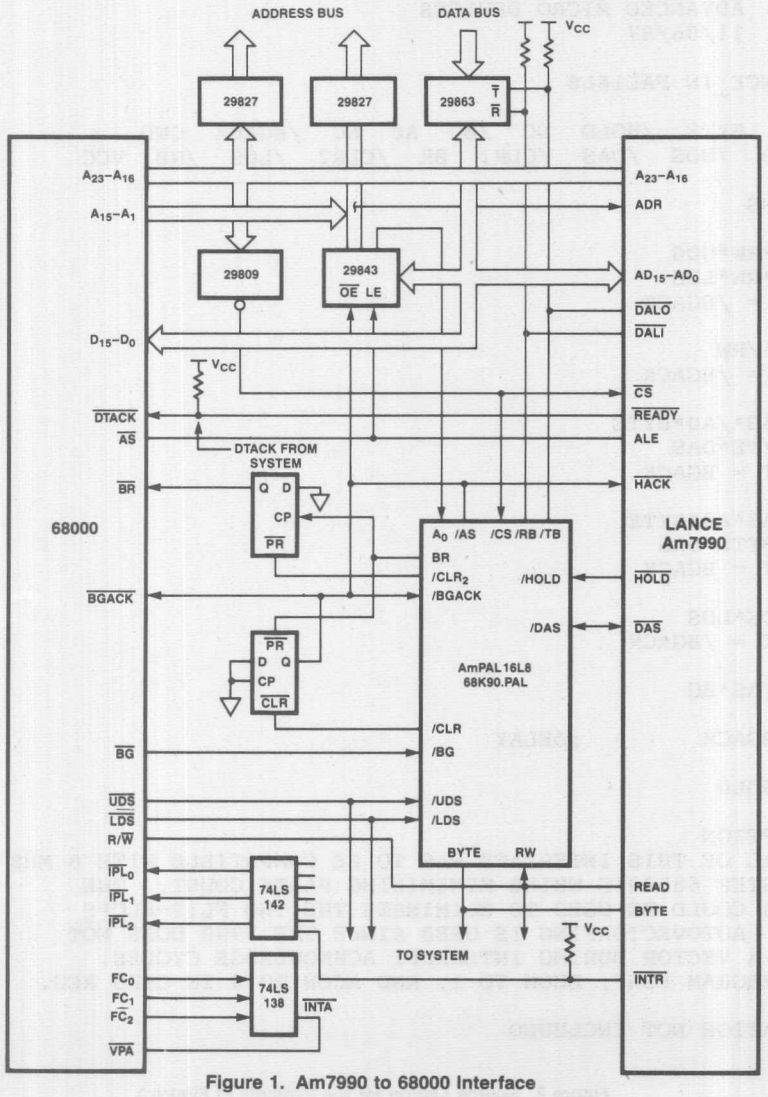


Figure 1. Am7990 to 68000 Interface

Autovectoring is used since the Am7990 does not return a vector during interrupt acknowledge cycles. The BYTE and $\overline{DAS}$ signals of LANCE are used to generate the $\overline{UDS}$ and $\overline{LDS}$ when LANCE is in Bus Master mode; the $\overline{UDS}$ and $\overline{LDS}$ is used to generate the $\overline{DAS}$ when LANCE is in Bus Slave mode. It takes two latches to demultiplex the LANCE address/data lines to adapt to the 68000 address bus. The flip-flops can be replaced by a PAL22V10 to minimize parts count. Equations for the PAL16L8 are shown in Figure 2.

```
TITLE     68000 TO LANCE INTERFACE
PATTERN   PAT002
REVISION  01
AUTHOR    JOE BRCICH
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP LANCE_IN PAL16L8

/AS   RW   BYTE   /HOLD   NC   /BG   A0   NC   /BGACK   GND
/CS   /TB   /UDS   /DAS   /CLR1   BR   /CLR2   /LDS   /RB   VCC

EQUATIONS

RB = CS*RW*UDS
   + CS*RW*LDS
RB.TRST = /BGACK

TB = CS*/RW
TB.TRST = /BGACK

UDS = DAS*/A0*BYTE
    + BYTE*DAS
UDS.TRST = BGACK

LDS = DAS*A0*BYTE
    + /BYTE*DAS
LDS.TRST = BGACK

DAS = UDS*LDS
DAS.TRST = /BGACK

CLR1 = /AS*BG

CLR2 = BGACK          ;DELAY

/BR = /HOLD

;DESCRIPTION
;THE GOAL OF THIS INTERFACE WAS TO BE COMPATIBLE WITH 8 MHZ
;AND FASTER 68000'S WHILE MINIMIZING PARTS COUNT.  THE
;Am22V10 COULD BE USED TO ELIMINATE THE TWO FLIP-FLOPS
;SHOWN.  AUTOVECTORFING IS USED SINCE THE 7990 DOES NOT
;RETURN A VECTOR DURING INTERRUPT ACKNOWLEDGE CYCLES.
;NOTE PROGRAM BSWP, BCON TO 1, AND ACON TO 0 IN CSR3 REG.

; SIMULATION NOT INCLUDED
```

Figure 2. Source Listing for the Example of Figure 1

68000 to AmZ8068 Data Ciphering Processor Interface

Figures 1 and 2 show the 68000-DCP interface and the interface timing. This interface provides a two-chip solution to add high-speed data ciphering to a 68000-based system. About 500 kbyte/sec are possible in a CPU-controlled transfer. The ciphering rate can be increased with a sophisticated DMA controller, or with several DCPs operating in parallel. The CPU operates at 8 MHz and the DCP operates synchronously at 4 MHz. The Interface Controller, a PAL device, generates the Address and Data Strobes for the DCP and the Data Acknowledge for the CPU. It also divides the CPU clock by two and synchronizes it to the Data Strobes.

The main features of this interface are:

— Multiplexed Control Mode

— Demultiplexed address and data bus

— Two-Cycle Operation

— Clock Synchronization with two Low Cycles after the Data Strobes

— About 500 kbyte/sec ciphering speed

Data transfers between the CPU and the DCP are accomplished by a two-cycle operation. First the address of an internal register is latched in, then the data is transferred. This causes a small overhead in the initialization phase, but improves the ciphering rate in a high-speed data ciphering session. The rate of 500 kbyte/sec can be reached only if a high-speed peripheral device is connected to the Slave Port and the DCP is programmed for dual port configuration.

The PAL device is programmed to allow only DCP transfers to the DCP. The PAL device equations are shown in Figure 5. A_0 must be odd to make the CPU transfer the data on the Low byte of the data bus. A "0" on A_1 indicates an Address Latch Cycle, whereas a "1" on A_1 indicates a Data Transfer Cycle. A_0 must be "1" in both cycles.

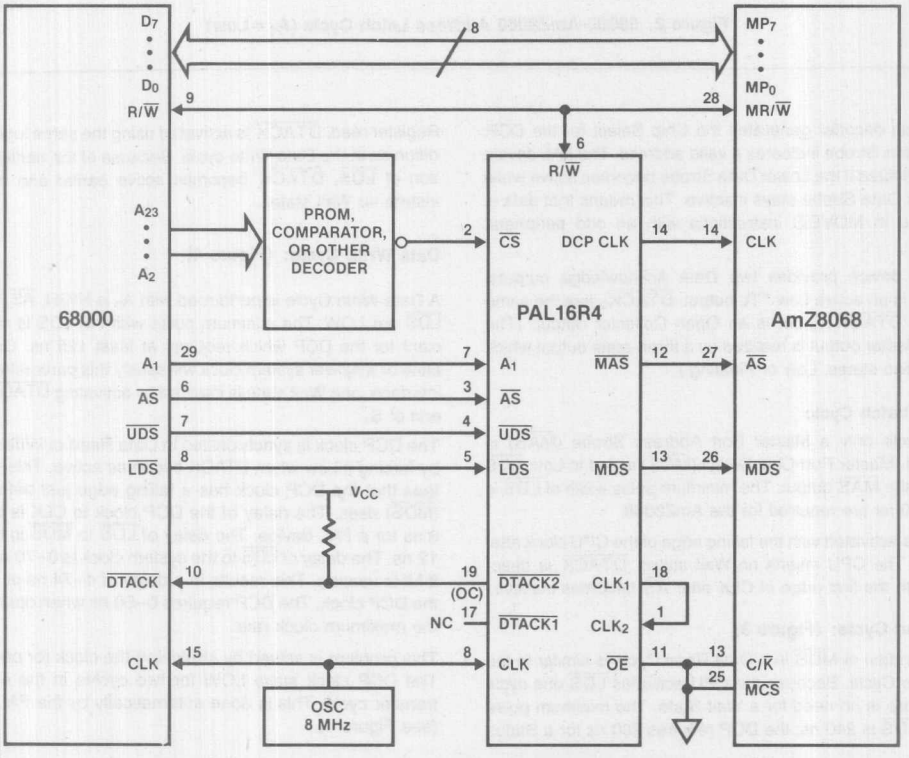
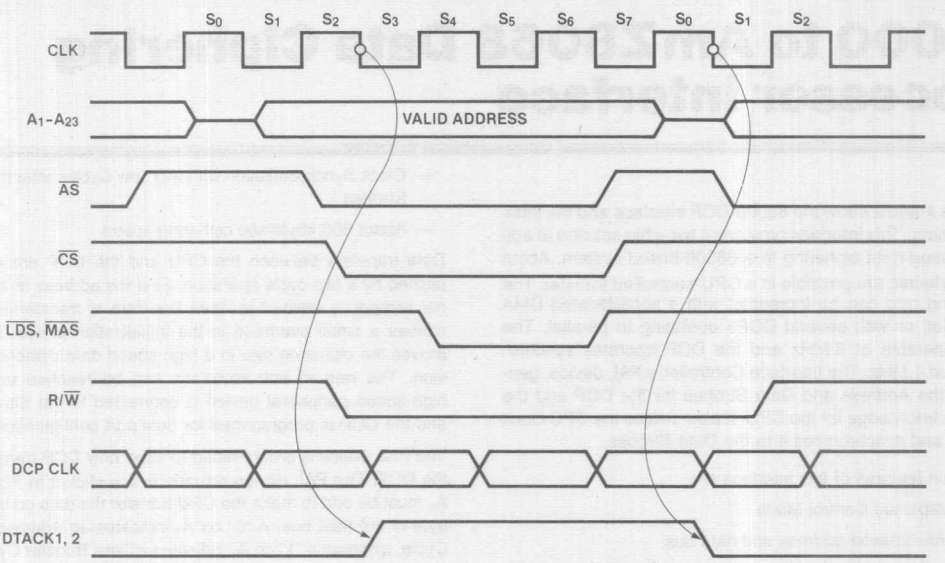


Figure 1. AmZ8068 to 68000 Interface

02188A-73



02188A-74

Figure 2. 68000-AmZ8068 Address Latch Cycle (A_1 = Low)

An address decoder generates the Chip Select for the DCP. The Address Strobe indicates a valid address. The PAL device is only activated if the Lower Data Strobe becomes active while the Upper Data Strobe stays inactive. This means that data is transferred in MOVE.B instructions with an odd peripheral address.

The PAL device provides two Data Acknowledge outputs. $\overline{DTACK}_1$ is an active Low TTL output. $\overline{DTACK}_2$ has the same timing as $\overline{DTACK}_1$, but is an Open Collector output. (The Open Collector output is realized by a three-state output which has only two states, Low or Floating.)

Address Latch Cycle

In this cycle only a Master Port Address Strobe ($\overline{MAS}$) is generated. Master Port Chip Select ($\overline{MCS}$) is tied to Low. $\overline{LDS}$ is sent to the $\overline{MAS}$ output. The minimum pulse width of $\overline{LDS}$ is 115 ns; 80 ns are required for the AmZ8068.

$\overline{DTACK}$ is activated with the falling edge of the CPU clock after cycle S_2. The CPU inserts no Wait states. $\overline{DTACK}$ is deactivated with the first edge of CLK after $\overline{AS}$ becomes inactive.

Data Read Cycle: (Figure 3)

The generation of $\overline{MDS}$ in a Data Read Cycle is similar to the Data Write Cycle. Because the CPU activates $\overline{LDS}$ one cycle earlier, there is no need for a Wait State. The minimum pulse width of $\overline{LDS}$ is 240 ns; the DCP requires 200 ns for a Status Register read. $\overline{DTACK}$ is activated using the same logical condition as in the Data Write cycle. Because of the earlier activation of $\overline{LDS}$, $\overline{DTACK}$ becomes active earlier and the CPU inserts no Wait states.

Data Write Cycle: (Figure 4)

A Data Write Cycle is performed with A_0 is HIGH, $\overline{AS}$, $\overline{CS}$ and $\overline{LDS}$ are LOW. The minimum pulse width of LDS is not sufficient for the DCP which requires at least 125 ns. One Wait state or a slower system clock will satisfy this parameter. In this interface, one Wait state is inserted by activating $\overline{DTACK}$ at the end of S_4.

The DCP clock is synchronized in Data Read or Write Cycles by forcing it Low when DTACK becomes active. This guarantees that the DCP clock has a falling edge just before $\overline{LDS}$ ($\overline{MDS}$) rises. The delay of the DCP clock to CLK is typically 8 ns for a PAL device. The delay of $\overline{LDS}$ to $\overline{MDS}$ is typically 12 ns. The delay of $\overline{LDS}$ to the system clock is 0–70 ns for the 8 MHz version. This results in a delay of 4–74 ns of $\overline{MDS}$ to the DCP clock. The DCP requires 0–50 ns when operating at the maximum clock rate.

This problem is solved by stretching the clock for one cycle. The DCP clock stays LOW for two cycles in the end of a transfer cycle. This is done automatically by the PAL device (see Figure 4).

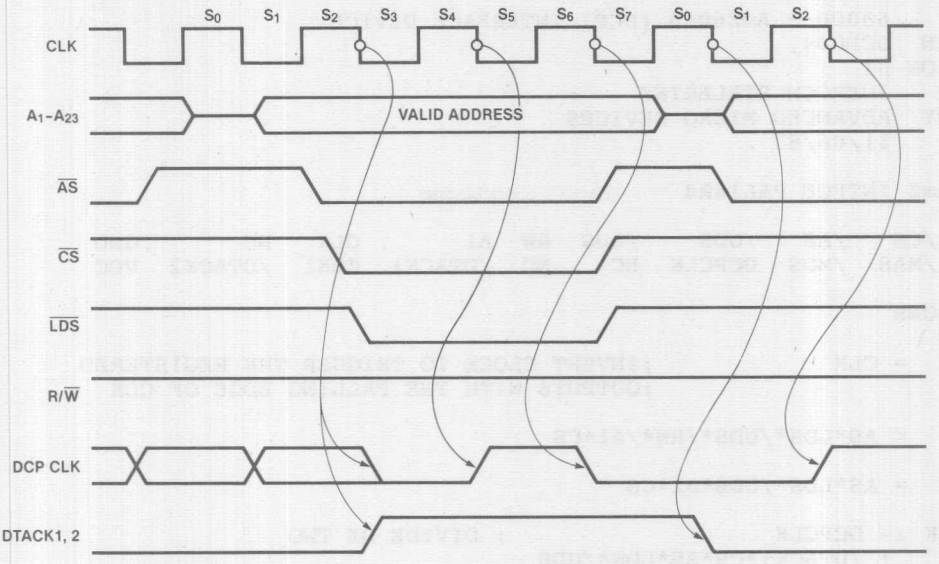


Figure 3. 68000-AmZ8068 Data Read Cycle (A_1 = HIGH)

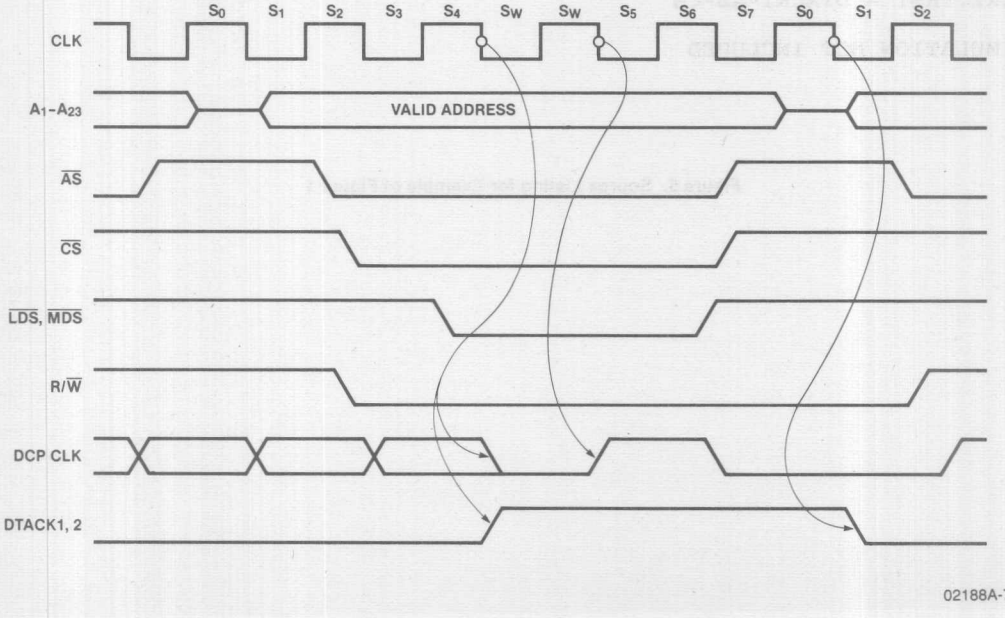


Figure 4. 68000-AmZ8068 Data Write Cycle (A_1 = HIGH)

```
TITLE    68000 - AmZ8068 (DCP) INTERFACE DEVICE
PATTERN  DCP044
REVISION 01
AUTHOR   JUERGEN STELBRINK
COMPANY  ADVANCED MICRO DEVICES
DATE     11/06/87

CHIP AmZ_INTFCE PAL16R4

CLK2  /CS   /AS   /UDS    /LDS  RW  A1       CLK   NC       GND
/OE   /MAS  /MDS  DCPCLK  NC    NC  /DTACK1  CLK1  /DTACK2  VCC

EQUATIONS

/CLK1   = CLK              ;INVERT CLOCK TO TRIGGER THE REGISTERED
                           ;OUTPUTS WITH THE FALLING EDGE OF CLK

MAS     = AS*LDS*/UDS*/RW*/A1*CS

MDS     = AS*LDS*/UDS*A1*CS

/DCPCLK := DCPCLK                      ; DIVIDE BY TWO
         + /DTACK1*CS*AS*LDS*/UDS
         + DTACK1*/AS*/LDS*/UDS        ; TWO CLOCKS LOW IN
                                       ; THE END OF A DATA CYCLE

DTACK1  := AS*LDS*/UDS*A1*CS           ; DATA TRANSFER CYCLE
         + AS*/RW*/A1*CS               ; ADDRESS LATCH CYCLE

DTACK2 = DTACK1

DTACK2.TRST = DTACK1*AS*CS

; SIMULATION NOT INCLUDED
```

Figure 5. Source Listing for Example of Figure 1

68000 and Dual Am9516 DMA Controllers Interface

There has been interest shown in connecting two Am9516 DMA Controllers to obtain four channels. The example here shows that such a system can be built by incorporating one PAL device, AMD's 22V10 (Figure 1). Address and data buses are not shown as they are straightforward and require no explanation. The PAL device, designated 68K16D2, converts the two DREQs into the 68000 three-wire handshake, prioritizes the request, and converts the control signals appropriately. Equations for the PAL device are shown in Figure 2.

The key parameters are: 1) data hold with respect to the rising edge of $\overline{DS}$ during a write, and 2) $\overline{DTACK}$ setup time. Control for a data bus transceiver is shown because it will be required in most systems. The PAL device provides these signals when the CPU is bus master; the Am9516 generates these control signals directly when it is bus master.

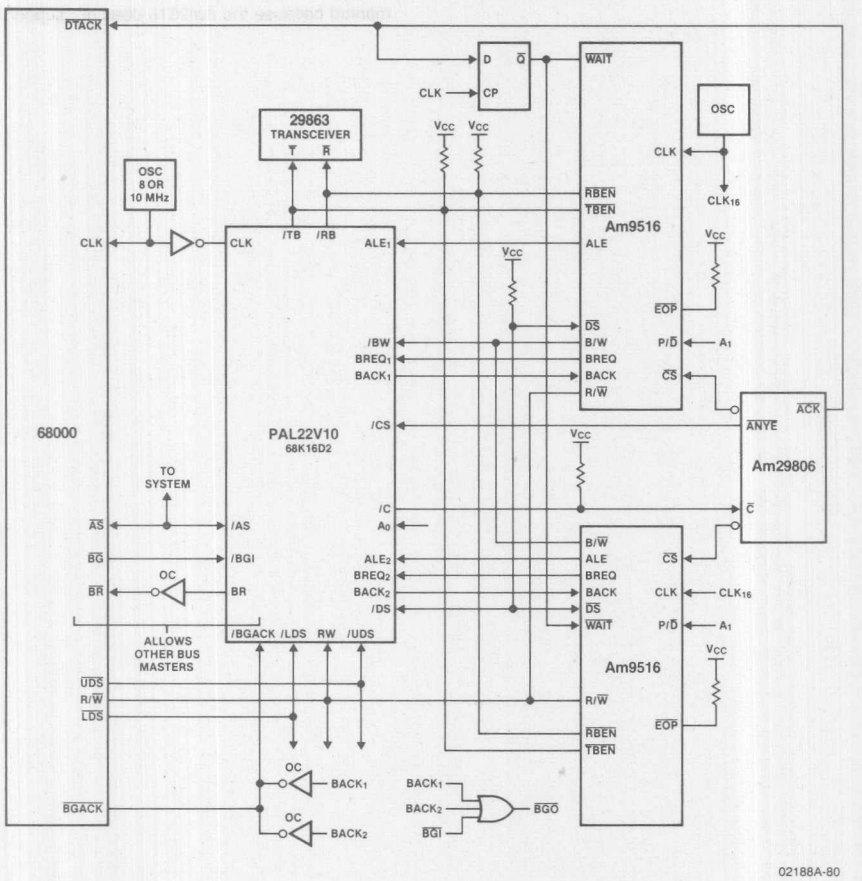


Figure 1. Dual Am9516 UDCs to 68000 CPU Interface

The Am9516s are shown with independent clocks. The clocks may be divided from the CPU clock or may be generated independently of the CPU. Because $\overline{WAIT}$ must meet the set-up and hold times, $\overline{DTACK}$ may need synchronization by the use of a flip-flop, as shown. This flip-flop may be eliminated in synchronous systems.

The clock is inverted to the PAL device to meet $\overline{DTACK}$ set-up time in systems of 10 MHz or faster. This inverter may be deleted in slower systems, if appropriate changes to the PAL device equations are made. This PAL device implements fixed priority between the Am9516s. $BREQ_1$ is the highest priority. Rotating priority can be implemented by adding another PAL device.

$\overline{EOP}$s are pulled up separately, but could be tied together since they affect a channel only if it is active. The bus-error function can be supported by connecting BERR and EOP.

If a bus error occurs, $\overline{EOP}$ will stop the current transfer and interrupt the CPU. The Interrupt Service routine can read the status to determine if $\overline{EOP}$ caused the interrupt or if termination was normal. If $\overline{EOP}$ caused the interrupt, the Address Register can be read to determine where the bus error occurred. After the problem is corrected, the CPU can program the Am9516 to complete the transfer or do an alternate transfer, as appropriate.

When operating the DMA in interleave mode, an external $\overline{EOP}$ should be gated with $\overline{DACK}$ to prevent affecting the wrong channel. This is unnecessary if interleave is not used, since the UDC releases the bus.

This arbiter design supports both serial and parallel-expansion techniques and is therefore compatible with VME bus protocol. Bus grant out was implemented with an external gate due to a shortage of pins. The VME/BCLR function was not implemented because the Am9516 does not support pre-emption.

```
TITLE     68000 TO DUAL 9516 INTERFACE
PATTERN   PAT001
REVISION  01
AUTHOR    JOE BRCICH
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP DUAL_9516 PAL22V10

CLK   RW   A0   BREQ1   BREQ2    /BG   NC   ALE1   /BW   ALE2   /BGACK   GND
/CS   /LDS    /UDS   /DS   /C    /AS   /BR   BACK2   BACK1   /TB   /RB   VCC

EQUATIONS

BR = BREQ1*/BGACK
   + BREQ1*BG
   + BREQ2*/BGACK
   + BREQ2*BG

BACK1 = BREQ1*BG*/AS
      + /BG*BGACK

BACK2 = BREQ2*/BREQ1*BG*/AS
      + /BG*BGACK

RB = CS*RW*UDS
   + CS*RW*LDS
RB.TRST = /BGACK

TB = CS*/RW
TB.TRST = /BGACK

DS = AS*/C*/RW
   + AS*RW
DS.TRST = /BGACK*AS

AS = ALE1
   + ALE2
AS.TRST = BGACK

UDS = DS*/A0*/BW
     + BW*DS
UDS.TRST = BGACK

LDS = DS*A0*/BW
     + BW*DS
LDS.TRST = BGACK

C := UDS*BGACK
   + LDS*BGACK
C.TRST = AS
```

Figure 2. Source Listing for the Example of Figure 1

Am8530 to 68020 Interface

This design shows the logic to interface a 6 MHz Am8530 Serial Communications Controller to the 68020 CPU (Figure 1) running at 10 MHz with a system clock cycle time of 100 ns. The Am8530 is a high-performance, dual-channel SCC that supports data rates up to 1.5M bps and a variety of communication protocols.

The PAL device generates the /RD and /WR control signals for the Am8530 from the 68020 /DS and R/W- control signals. It also generates the clock for the SCC by dividing the 68020 system clock. This meets the 165-ns minimum cycle time for the 6-MHz SCC clock.

Also, a state machine is implemented to perform the handshake necessary for byte transfers on the 68020 bus. The state machine transition diagram is shown in Figure 2. Normally, DSACK[1:0] is inactive. When this device is selected, a wait state is inserted and then data transfer is acknowledged by asserting DSACK[1:0] for byte transfer. When the CPU deasserts the address strobe, DSACK[1:0] is negated. DSACK[1:0] will be driven by other devices on the 68020 system bus. The PAL device enables DSACK[1:0] only when the CPU performs a read or write cycle for the SCC; at other times DSACK[1:0] lines are three-stated.

Since the Am8530 is a byte-wide peripheral, it is connected to the upper byte of the 32-bit wide 68020 data bus.

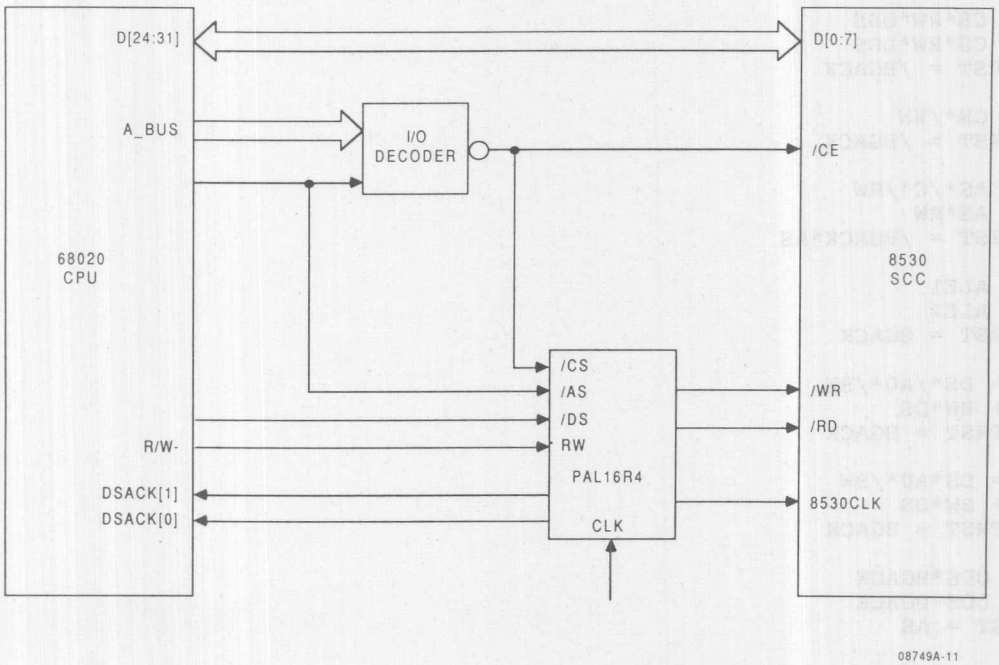


Figure 1. Am8530 to 68020 Interface

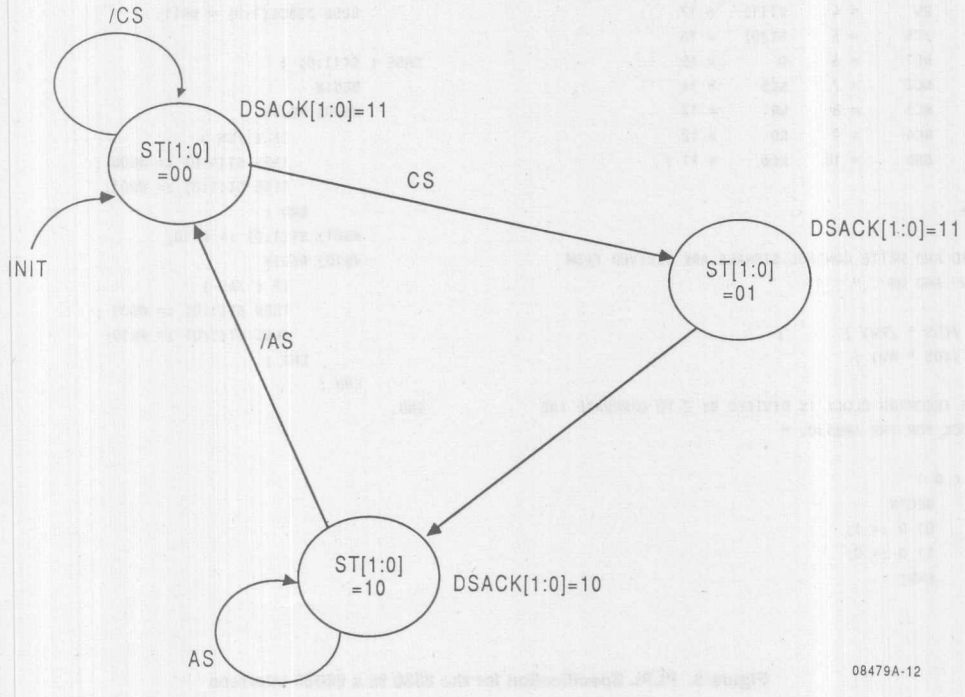


Figure 2. 68020 Byte Port DSACK Handshake State Diagram

```
" THIS PLPL FILE IS FOR A 16R4 THAT IMPLEMENTS THE LOGIC
  NECESSARY TO INTERFACE AN Am8530 (SCC) TO A 68020 SYSTEM. "

DEVICE Am8530_TO_68020 (PAL16R4)

PIN
        CLK     = 1     VCC      = 20
        /AS     = 2     DSACK[1] = 19
        /DS     = 3     DSACK[0] = 18
        RW      = 4     ST[1]    = 17
        /CS     = 5     ST[0]    = 16
        NC1     = 6     Q        = 15
        NC2     = 7     NC5      = 14
        NC3     = 8     WR       = 13
        NC4     = 9     RD       = 12
        GND     = 10    NC6      = 11 ;

BEGIN

" READ AND WRITE CONTROL SIGNALS ARE DERIVED FROM
  R/W- AND DS-. "

WR = /(DS * /RW) ;
RD = /(DS * RW) ;

" THE INCOMING CLOCK IS DIVIDED BY 2 TO GENERATE THE
  CLOCK FOR THE Am8530. "

CASE ( Q )
        BEGIN
        0) Q := 1;
        1) Q := 0;
        END;

" THE FOLLOWING CODE IMPLEMENTS THE STATE MACHINE TO
  PERFORM DSACK OPERATION. IT INSERTS WAIT STATES,
  ASSERTS DSACK FOR BYTE TRANSFER AND REMOVES DSACK
  WHEN AS IS NEGATED. "

IF ( /CS ) ENABLE ( DSACK[1:0] )

IF ( ST[1:0] = #B10 )
        THEN DSACK[1:0] = #B10;
        ELSE DSACK[1:0] = #B11;

CASE ( ST[1:0] )
        BEGIN
        #B00) BEGIN
                IF ( /CS )
                THEN ST[1:0] := #B00 ;
                ELSE ST[1:0] := #B01;
              END ;
        #B01) ST[1:0] := #B10;
        #B10) BEGIN
                IF ( /AS )
                THEN ST[1:0] := #B00 ;
                ELSE ST[1:0] := #B10;
              END ;
        END ;
END.
```

Figure 3. PLPL Specification for the 8530 to a 68020 Interface

Interfacing to the 8088

Overview

The 8088 CPU is an 8-bit processor designed around the 8086 internal structure. Most functions of the 8088 are identical to the equivalent 8086. The 8088 fetches and writes 16-bit words in two consecutive bus cycles. Both the 8086 and the 8088 handle the external bus the same way but the 8088 handles only 8-bits at a time; and both appear identical to the software engineer, with the exception of execution time.

The hardware interface of the 8088 contains the major differences between the two CPUs. The pin assignments are nearly identical, however, with the following functional changes:

A_8-A_{15}	These pins are only address outputs on the 8088. They are latched internally and remain valid throughout a bus cycle in a manner similar to the 8085 upper address lines.
$\overline{SS0}$	Provides the $\overline{SO}$ status information in the minimum mode. This output occurs on pin 34 in minimum mode only.
DT/$\overline{R}$, IO/$\overline{M}$,	and $\overline{SS0}$ provide the complete bus status in minimum mode.
IO/$\overline{M}$	has been inverted to be compatible with the MCS-85 bus structure.

ALE is delayed by one clock cycle in the minimum mode when entering HALT, to allow the status to be latched with ALE.

The 8088 and AMD Proprietary Peripherals

The evolution of chip design has taken the 8-bit environment into the 16-bit environment. While the new generation of peripheral devices are often 16 bits wide, the older, established 8-bit orientation of CPUs and peripherals are still significant. Interfacing a 16-bit peripheral with an 8-bit CPU often encounters data path incompatibility and involves bus control manipulation. This type of integration mainly involves separating the control and data paths from the new peripheral and the system.

The ability to mix different data path widths can improve system functionality, performance, and cost. It is less expensive to use an 8-bit bus in a new design because the memory requirements are generally cheaper. A designer can use this data path mixing to upgrade the existing system until a new system design is warranted, or the designer can simply improve on the existing design as new peripherals become available.

8088 to Am9516 UDC Interface

Figure 1 shows the Data Bus and Control Interface between an 8088 microprocessor and an Am9516. This interface is accomplished by using an AmPAL22V10, a 74LS161 counter, an Am2952 bidirectional I/O port, and an Am2949 bidirectional transceiver. Figures 2 and 3 show the PAL device timing for both the Bus Master and the Bus Slave cases.

In this design, certain simplifying constraints have been made. Word operations are only allowed during command chaining operations when the Am9516 is Bus Master. During Slave Write operations, the first byte output to the Am9516 must have an odd address, and the following second byte an even address. Conversely, during a Slave Read cycle, the first byte read from the Am9515 must be at an even address and the second at the next higher odd address. Furthermore, for both slave and master operations, the system must use the latched A_O (LA_O) from the PAL22V10 as its sole A_O. That is, this LA_O must circumvent any address latching to memory and so is to be used directly.

As can be seen from the figures, the AmPAL22V10 manipulates the control signals to the transceivers and latches so that bytes can be funneled into words during Am9516 Command Chaining operations and Slave Writes, and words can be funneled into bytes during Am9516 Slave Reads. Also, LAO is alternately toggled in order to provide the dual address (16-bits) during chaining operations.

Figure 4 shows the PLPL equations and resulting sum-of-products equations for the design (PLPL is a higher level "C-like" language for PAL devices). For those not familiar with "C" or Pascal, the intermediate sum-of-products form shows the equations in Boolean form.

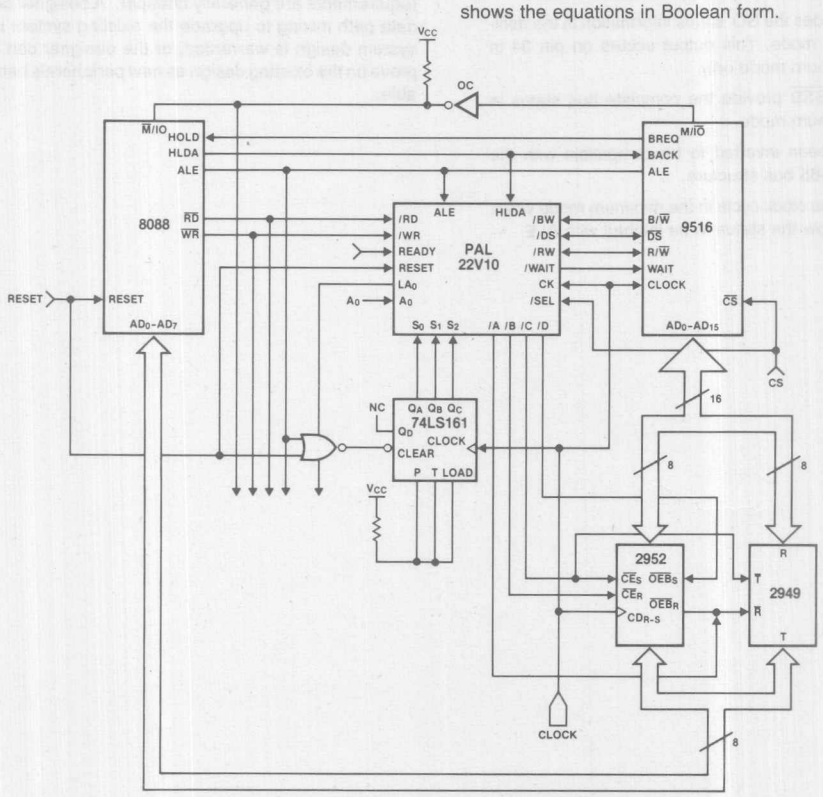
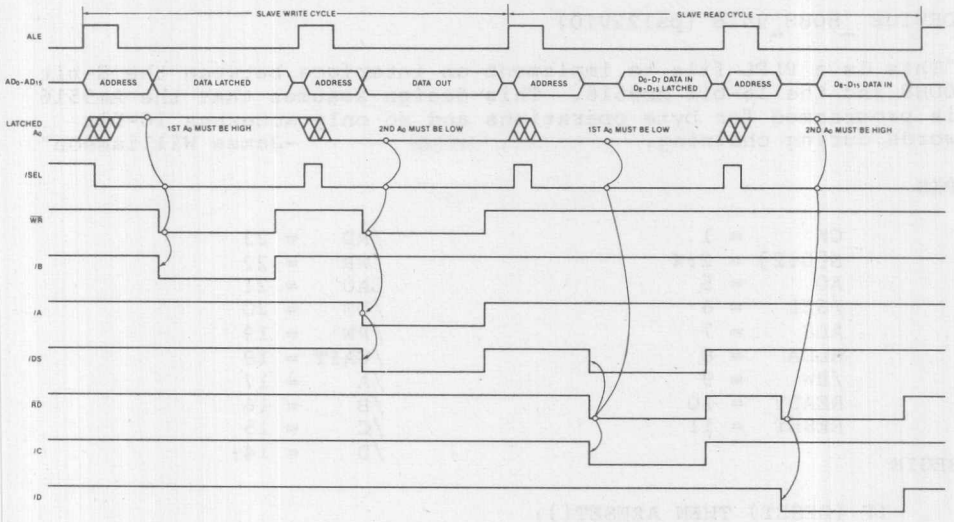


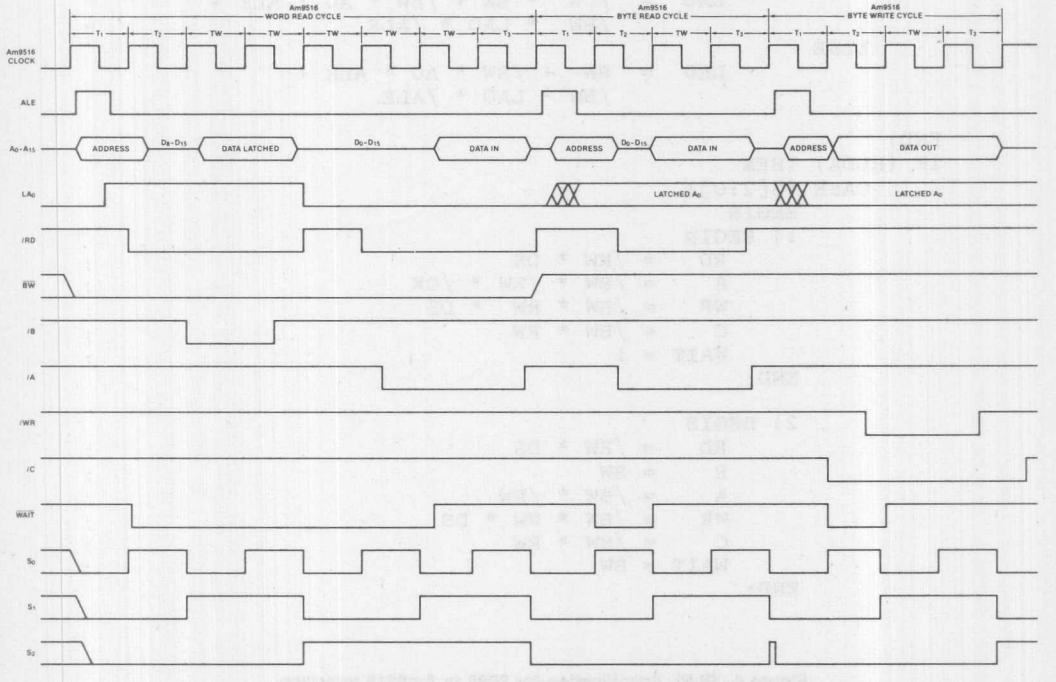
Figure 1. The Am9516 UDC to 8088 CPU Data Bus and Control Interface

02188A-60



02188A-61

Figure 2. The Am9516 UDC to 8088 CPU Bus Slave Timing



02188A-62

Figure 3. The Am9516 UDC to 8088 CPU Bus Master Timing

```
DEVICE _8088_9516 (pal22V10)

"This is a PLPL file to implement an interface between the 8-bit
8088 and the 16-bit Am9516.  This design assumes that the Am9516
is programmed for byte operations and so only accesses 16-bit
words during chaining.                          -James Williamson"

PIN

        CK      = 1                     /RD    = 23
        S[0:2]  = 2:4                   /WR    = 22
        A0      = 5                     LA0    = 21
        /SEL    = 6                     /DS    = 20
        ALE     = 7                     /RW    = 19
        HLDA    = 8                     /WAIT  = 18
        /BW     = 9                     /A     = 17
        READY   = 10                    /B     = 16
        RESET   = 11                    /C     = 15
                                        /D     = 14;
BEGIN

     IF (RESET) THEN ARESET();
"============================================================="
"This section defines the wiggles when the Am9516 is Bus Master"
"============================================================="
     IF (HLDA)  THEN ENABLE();
     IF (/S[2] * HLDA) THEN BEGIN
          IF (/S[1] * /S[0]) THEN
                    LA0  =  /CK  * BW + /BW * A0 * ALE +
                            /BW  * LA0 * /ALE                ;
          ELSE
                    LA0  =  BW  + /BW * A0 * ALE +
                            /BW * LA0 * /ALE                 ;

     END;
     IF (HLDA) THEN
          CASE (S[2:0])
               BEGIN
               1) BEGIN
                    RD   = /RW * DS                          ;
                    A    = /BW * /RW * /CK                   ;
                    WR   = /BW * RW  * DS                    ;
                    C    = /BW * RW                          ;
                    WAIT = 1                                 ;
               END;

               2) BEGIN
                    RD   = /RW * DS                          ;
                    B    = BW                                ;
                    A    = /BW * /RW                         ;
                    WR   = /BW * RW * DS                     ;
                    C    = /BW * RW                          ;
                    WAIT = BW                                ;
               END;
```

Figure 4. PLPL Specification for 8088 to Am9516 Interface

```
            3) BEGIN
                 RD   = /RW * DS * B                    ;
                 B    = BW * CK                         ;
                 A    = /BW * RD                        ;
                 WR   = /BW * RW * DS                   ;
                 C    = /BW * RW                        ;
                 WAIT = BW                              ;
            END;

            5) BEGIN
                 RD   = /RW * DS                        ;
                 A    = /BW * /CK                       ;
                 WAIT = BW                              ;
            END;

            6) BEGIN
                 RD   = /RW * DS                        ;
                 A    = BW                              ;
            END;

            7) BEGIN
                 RD   = /RW * DS                        ;
                 A    = /RD                             ;
            END;

       END;

"================================================================="
"This  section  defines the wiggles when the 8088 is Bus  Master"
"================================================================="

    BEGIN

       LA0 = A0 * ALE * SEL +
             LA0 * /ALE * SEL                           ;
       B   = LA0 * WR * SEL                             ;
       A   = /LA0 * WR * SEL                            ;
       DS  = A +
             /LA0 * RD * SEL                            ;
       C   = /LA0 * RD * SEL                            ;
       D   = LA0 * RD * SEL                             ;
    END;

END.
```

Figure 4. PLPL Specification for 8088 to Am9516 Interface (Continued)

80186 to Am9516 Universal DMA Controller Interface

The addition of the Am9516 to an 80186 design is a natural choice in systems requiring four channels of DMA. Figure 1 shows the interface between the 80186 and the Am9516 with a PAL device. PCS_5 is programmed to provide a latched A_1 signal.

This interface accomplishes two major control transformations. First, it converts $\overline{RD}$ and $\overline{WR}$ into R/W and $\overline{DS}$ when the 80186 is Bus Master, and vice versa when the Am9516 is Bus Master. Secondly, the transceiver control signals, $\overline{TBEN}$ and $\overline{RBEN}$, are generated from DEN and DT/R. This example shows only one possible configuration. Other configurations can be made as dictated by system requirements. A PAL device is used here to reduce board space.

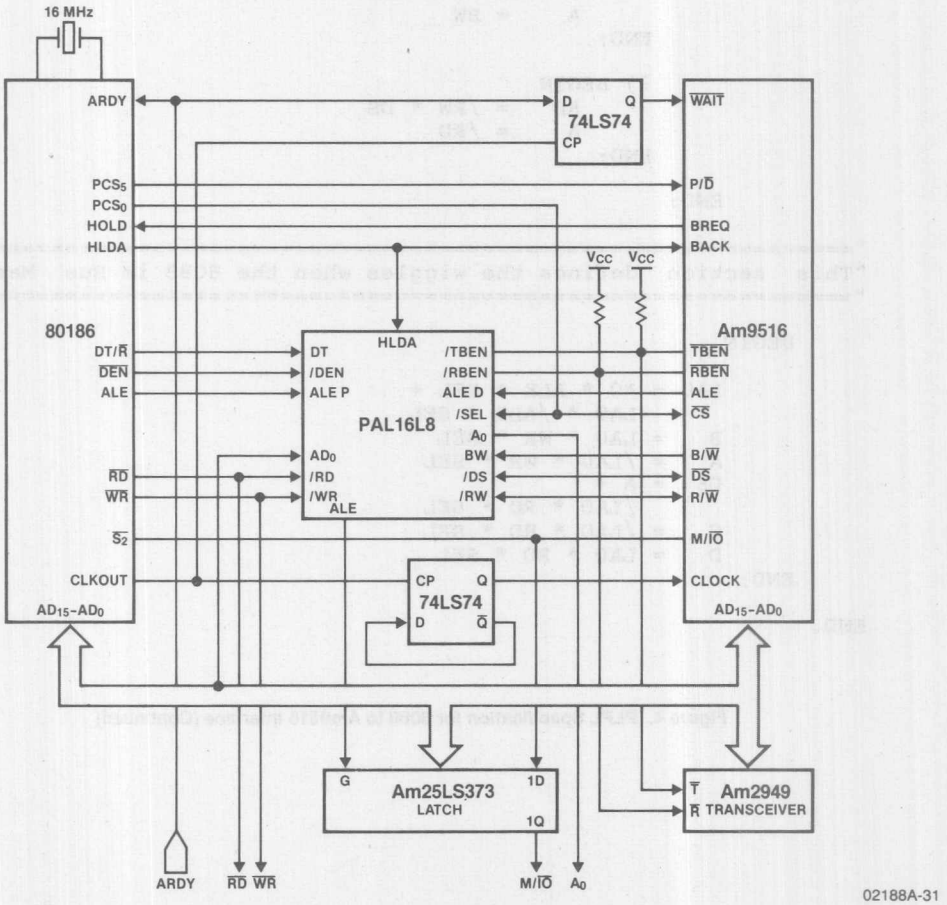


Figure 1. Am9516 to 80186 Interface

```
TITLE     Am9516 TO 80186 INTERFACE
PATTERN   PAT001
REVISION  01
AUTHOR    JOE ENGINEER
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP DMA_INT PAL16L8

NC ALED ALEP HLDA BW AD0 DT /DEN /SEL GND
NC /RBEN /RD ALE A0 /RW /DS /WR /TBEN VCC

EQUATIONS

DS = RD + WR
DS.TRST = /HLDA

RW = DT
RW.TRST = /HLDA

TBEN = /DT * /SEL * DEN
TBEN.TRST = /HLDA

RBEN = DT * /SEL * DEN
RBEN.TRST = /HLDA

RD = /RW * DS
RD.TRST = HLDA

WR = RW * DS
WR.TRST = HLDA

ALE = /ALEP * /ALED

A0 = /AD0 * /BW * HLDA * ALED
   +  AD0 * BW * HLDA * ALED
   + /AD0 * /HLDA * ALEP
   +  A0 * /ALEP
   +  A0 * /ALED

; DESCRIPTION

; THIS PAL DEVICE CONVERTS THE CONTROL SIGNALS TO INTERFACE THE 80186
; TO THE Am9516 DMA CONTROLLER.

; SIMULATION NOT INCLUDED
```

Figure 2. Source Listing for Example of Figure 1

68000 Interrupt Controller

Introduction

Commercial and industrial microprocessor-based systems typically consist of a processor interfaced with many peripherals which randomly require service. To fully utilize the processor's computing potential, sources of hardware interrupts must be used to free the processor from software routines. The 68000 16-bit microprocessor has 256 different exception-processing vectors which point to predetermined locations in the program memory space. There are 192 external user interrupt vectors and seven autovector interrupts. Table 1 shows the exception vector assignments. The interrupt structure of the 68000 will be discussed in more detail along with two methods of designing an interrupt controller using PAL (Programmable Array Logic) devices.

Vector Number(s)	Address			Assignment
	Dec	Hex	Space	
0	0	000	SP	Reset: Initial SSP
—	4	004	SP	Reset: Initial PC
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12*	48	030	SD	(Unassigned, reserved)
13*	52	034	SD	(Unassigned, reserved)
14*	56	038	SD	(Unassigned, reserved)
15	60	03C	SD	Uninitialized Interrupt Vector
16-23*	64	04C	SD	(Unassigned, reserved)
	95	05F		—
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt Autovector
26	104	068	SD	Level 2 Interrupt Autovector
27	108	06C	SD	Level 3 Interrupt Autovector
28	112	070	SD	Level 4 Interrupt Autovector
29	116	074	SD	Level 5 Interrupt Autovector
30	120	078	SD	Level 6 Interrupt Autovector
31	124	07C	SD	Level 7 Interrupt Autovector
32-47	128	080	SD	TRAP Instruction Vectors
	191	0BF		—
48-63*	192	0C0	SD	(Unassigned, reserved)
	255	0FF		—
64-255	256	100	SD	User Interrupt Vectors
	1023	3FF		—

* Vector numbers 12, 13, 14, 16 through 23 and 48 through 63 are reserved for future enhancements. No user peripheral devices should be assigned these numbers.

Table 1. Exception Vector Assignment

68000 Exception Processing and Pin Description

The interrupt structure of the 68000 can grant up to 256 types of interrupt requests. These interrupts, or exceptions, are generated either by external or internal causes and are serviced by directing the flow of program to an exception processing routine. Table 1 lists these exceptions and their respective address in memory. Exception vectors are locations in memory where the processor fetches the address of a routine or subprogram which will service that exception. The exception vector is either internally or externally generated depending on the cause of the interrupt. The externally generated exceptions are interrupts which are placed on the interrupt control pins ($\overline{IPL2}$, $\overline{IPL1}$ and $\overline{IPL0}$), bus errors, and reset requests. The interrupts placed on control pins $\overline{IPL2}$-$\overline{IPL0}$ are requests from peripheral devices. The internally-generated exceptions come from instructions, address errors, or tracing. These different types of exceptions and their priorities are shown in Table 2. We will focus on group 1 interrupt exceptions.

Group	Exception	Processing
0	Reset Bus Error Address Error	Exception processing begins within two clock cycles.
1	Trace Interrupt Illegal Privilege	Exception processing begins before the next instruction.
2	TRAP, TRAPV, CHK, Zero Divide	Exception processing is started by normal instruction execution.

Table 2. Exception Grouping and Priority

The pinout of the 68000 is shown in Figure 1. The three interrupt control pins ($\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$) are asynchronous active low inputs which indicate the encoded priority level of

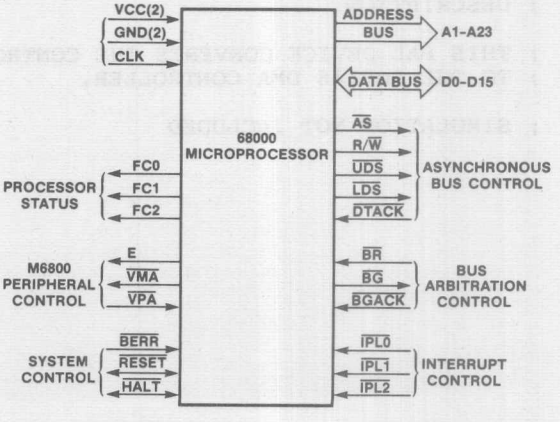


Figure 1.

the device requesting an interrupt. The least significant bit is $\overline{IPL0}$ and the most significant bit is $\overline{IPL2}$. Level seven, ($\overline{IPL2}$, $\overline{IPL1}$, $\overline{IPL0}$) = (000), has the highest priority, while level zero, (111), indicates that no interrupts are requested. All lower or equal priority interrupts are masked during the interrupt service routine of the present interrupting device. There are two ways by which a peripheral device can interrupt the normal flow of program: autovectoring or external vector number generation. The function code pins (FC2, FC1, and FC0) all become high during an interrupt acknowledge cycle. The interrupt acknowledge cycle always follows an interrupt request only after the present instruction cycle is completed. Thus interrupt acknowlege cycles come in between the instruction cycles and no information is lost.

The signals which are used during an external interrupt request are listed below with a description of their behavior and function.

ADDRESS BUS: the 24-bit address bus holds the address of the data to be accessed. During an interrupt acknowledge cycle the lower three bits (A3, A2, A1) hold the encoded level of the interrupt being serviced. If a level 5 interrupt is being serviced then (A3, A2, A1) will be (101) respectively.

DATA BUS: The lower eight bits of this 16-bit data bus must contain the vector number during a user interrupt acknowledge cycle. If an autovector routine is in process then the data bus is ignored.

$\overline{AS}$: The processor asserts address strobe anytime there is valid data on the address bus. It remains asserted for as long as the address is valid.

R/$\overline{W}$: This signal defines the data bus transfer as a read or a write cycle. During an interrupt acknowledge cycle the processor is in a read mode.

$\overline{UDS}$, $\overline{LDS}$: The upper and lower data strobes are asserted when the processor is in a read or write instruction cycle. Upper strobe enables the most significant byte of data while the lower strobe enables the least significant.

$\overline{DTACK}$: Data transfer acknowledge is an externally-generated signal which tells the processor that valid data is present on the data bus. If $\overline{DTACK}$ is not asserted before the falling edge of S4 (S4 is the fourth CPU clock state in a seven-state instruction cycle) then wait states are introduced.

$\overline{IPL2}$-$\overline{IPL0}$: The three interrupt control pins are inputs which contain the encoded priority level of the external interrupting device. Note that these are active low pins where (000) is level seven encoded.

FC2-FC0: Function code output pins indicate the processor's status. When they are all high the processor is in an interrupt acknowledge cycle.

E, $\overline{VMA}$, $\overline{VPA}$: Enable, valid memory address, and valid peripheral address are the standard 6800 family signals. $\overline{VPA}$ is also used when autovectoring.

The following two design examples use PAL units as a simple and cost-effective interrupt controller interface to the 68000. The first introduces external vector generation while the second shows how autovectoring can be done on the 68000.

2

Prioritized Individually Vectored Interrupt (Method 1)

As shown in the interrupt acknowledge sequence flow chart and timing diagram (Figure 2), a peripheral device must interrupt the processor by placing the encoded level of priority on the interrupt control pins ($\overline{IPL2}$-$\overline{IPL0}$). The processor then grants the interrupt after the completion of the current instruction cycle. The encoded priority level of the interrupt is placed on address bus bits A3-A1 during the interrupt acknowledge cycle. The status code lines (FC2-FC0) become high, indicating an interrupt acknowledge cycle. Once the lower data strobe ($\overline{LDS}$) is asserted the external device must place an 8-bit vector on the least significant byte of the data bus. This data is latched in state S7.

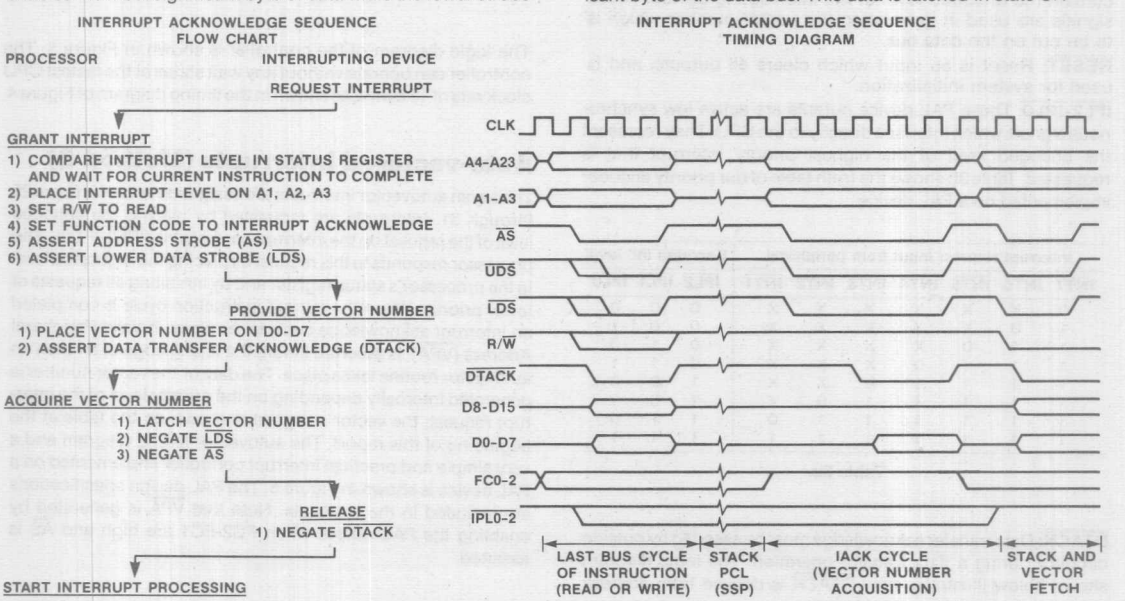


Figure 2.

As shown on Table 1 vectors 64-255 are assigned to the user interrupt vector numbers. These vectors must be generated externally and placed on the data bus during an interrupt acknowledge cycle. In our interrupt controller we arbitrarily choose vectors 249-255 as the vectors assigned to the interrupting peripheral devices. Table 3a shows this assignment and how the vector can be decoded from the address bits A1-A3.

Priority level	Vector assigned	Data								Address		
Device #	Decimal #	D7	D6	D5	D4	D3	D2	D1	D0	A3	A2	A1
1 (low priority)	249	1	1	1	1	1	0	0	1	0	0	1
2	250	1	1	1	1	1	0	1	0	0	1	0
3	251	1	1	1	1	1	0	1	1	0	1	1
4	252	1	1	1	1	1	1	0	0	1	0	0
5	253	1	1	1	1	1	1	0	1	1	0	1
6	254	1	1	1	1	1	1	1	0	1	1	0
7 (high priority)	255	1	1	1	1	1	1	1	1	1	1	1

Table 3a.

The two PAL devices used on this design example generate the interrupt vectors and all the necessary control signals. The various signals and their implementation on the PAL devices are explained below.

$\overline{INT7}$-$\overline{INT0}$: Any of the seven peripheral devices can request an interrupt by asserting one of these inputs. The interrupt must remain asserted until acknowledged by the CPU.

FC2-FC0 and $\overline{AS}$: Function code or processor status code become logical high during an interrupt acknowledge cycle. Address strobe is asserted anytime valid address is on the bus. $\overline{DTACK}$ and Data output control are decoded from these four outputs of the 68000.

A1-A3: The three least significant bits of the address bus contain the encoded level of the interrupting device. These signals are used in generating the vector number which is to be put on the data bus.

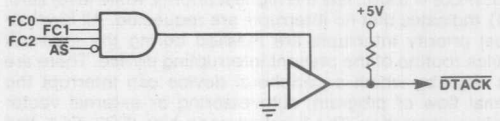
$\overline{RESET}$: Reset is an input which clears all outputs, and is used for system initialization.

$\overline{IPL2}$-$\overline{IPL0}$: These PAL device outputs are active low synchronous signals which interface directly to the CPU. They represent the encoded level of the highest priority interrupt that is requested. Table 3b shows the truth table of our priority encoder implemented on a PAL device.

Interrupt request input from peripheral							Encoded int. level		
$\overline{INT7}$	$\overline{INT6}$	$\overline{INT5}$	$\overline{INT4}$	$\overline{INT3}$	$\overline{INT2}$	$\overline{INT1}$	$\overline{IPL2}$	$\overline{IPL1}$	$\overline{IPL0}$
0	X	X	X	X	X	X	0	0	0
1	0	X	X	X	X	X	0	0	1
1	1	0	X	X	X	X	0	1	0
1	1	1	0	X	X	X	0	1	1
1	1	1	1	0	X	X	1	0	0
1	1	1	1	1	0	X	1	0	1
1	1	1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	1	1	1

Table 3b.

$\overline{DTACK}$: Data transfer acknowledge must be asserted by outside circuitry during a data transfer operation. The logic diagram shown below illustrates how $\overline{DTACK}$ is derived from address strobe and processor status signals.



D7-D0: The three least significant bits of the data output can be decoded straight from the address bits A3, A2 and A1. That is D2=A3, D1=A2 and D0=A1. The other five bits of data can be held high with pull-up resistors. Outputs of the three data bits become enabled by using the same scheme which enables the $\overline{DTACK}$ output.

$\overline{INTACK7}$-$\overline{INTACK1}$: Only one of these signals will be asserted during the interrupt acknowledge cycle. This signal feeds back into the interrupting device to tell that device that its interrupt has been acknowledged. We can use the 3-bit addresses to decode these signals as shown in Table 3c.

			$\overline{INTACK}$						
A3	A2	A1	7	6	5	4	3	2	1
0	0	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	0
0	1	0	1	1	1	1	1	0	1
0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	1	0	1	1	1
1	0	1	1	1	0	1	1	1	1
1	1	0	1	0	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1

Table 3c.

The logic diagram of the controller is shown in Figure 3. The controller can operate without any wait states at the fastest CPU clock rate of 12.5 MHz as shown in the timing diagram of Figure 4.

Auto-vectored Interrupts (Method 2)

The seven autovector interrupts are assigned vector numbers 25 through 31. Interrupts are requested by placing the encoded level of the request on the interrupt control pins ($\overline{IPL2}$-$\overline{IPL0}$). The processor responds to this request by placing the requested level in the processor's status register and by inhibiting all requests of lower priority. When the current instruction cycle is completed an interrupt acknowledge cycle takes place. If Valid Peripheral Address ($\overline{VPA}$), is asserted before the falling edge of S4 then an autovector routine takes place. The data or the vector number is generated internally depending on the priority level of the interrupt request; the vector assigned is shown on the table at the beginning of this report. The autovector timing diagram and a very simple and practical interrupt controller implemented on a PAL device is shown in Figure 5. The PAL design specifications are included in the appendix. Note that $\overline{VPA}$ is generated by enabling the PAL output when FC2-FC1 are high and $\overline{AS}$ is asserted.

Interrupt Controller Logic Diagram
External Vector Generation

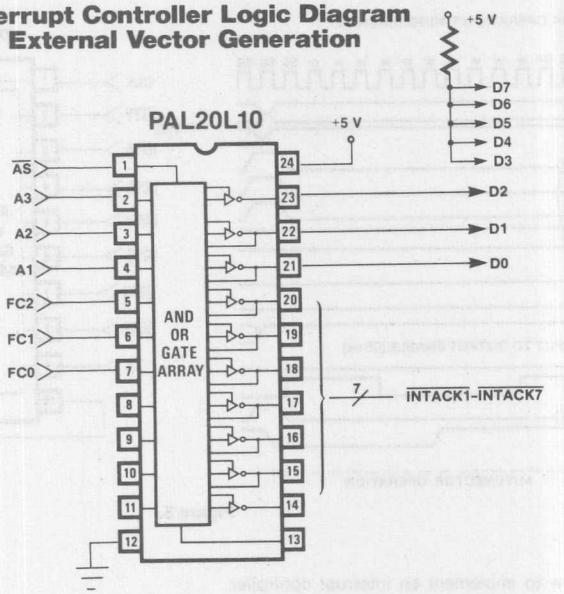
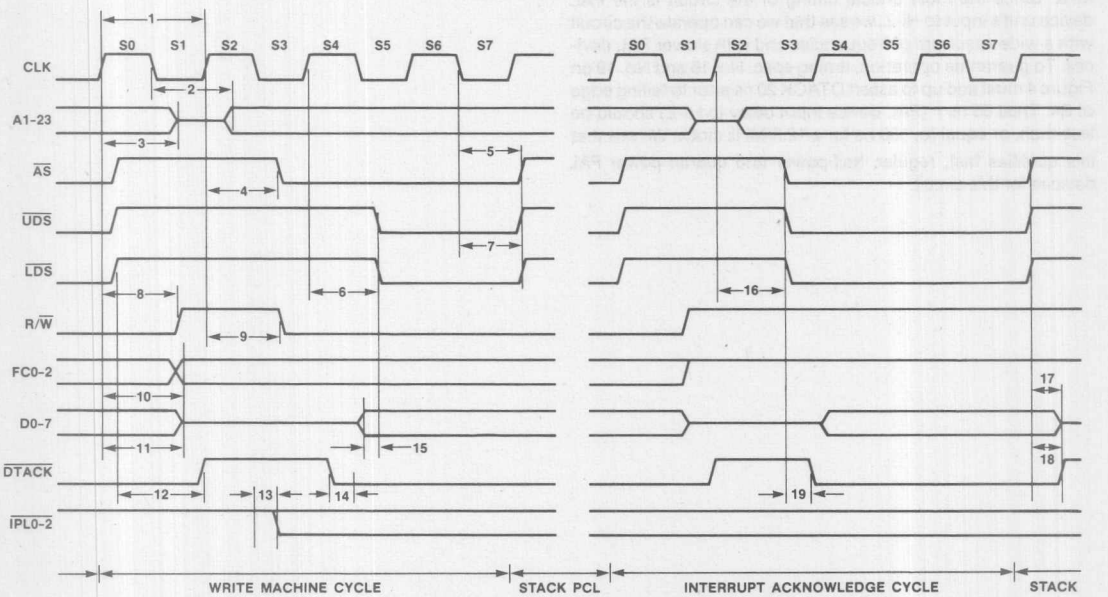


Figure 3.



1 CLOCK PERIOD (80 ns)
2 CLOCK LOW TO ADDRESS (55 ns)
3 CLOCK HIGH TO ADDRESS Hi-Z (60 ns)
4 CLOCK HIGH TO $\overline{AS}$ LOW (55 ns)
5 CLOCK LOW TO $\overline{AS}$ HIGH (50 ns)
6 CLOCK HIGH TO $\overline{DS}$ LOW (55 ns)
7 CLOCK LOW TO $\overline{DS}$ HIGH (50 ns)
8 CLOCK HIGH TO R/$\overline{W}$ HIGH (60 ns)
9 CLOCK HIGH TO R/$\overline{W}$ LOW (60 ns)
10 CLOCK HIGH TO FC VALID (55 ns)
11 CLOCK HIGH TO DATA Hi-Z (60 ns)
12 $\overline{AS}$ HIGH TO $\overline{DTACK}$ HIGH (70 ns)
13 PAL CLOCK TO OUT (25 ns)
14 MINIMUM SETUP TIME (20 ns)
15 DATA VALID TO $\overline{DS}$ LOW (15 ns)
16 CLOCK HIGH TO $\overline{AS}$, $\overline{DS}$ LOW (55 ns)
17 PAL INPUT TO Hi-Z (25 ns)
18 PAL INPUT TO Hi-Z (25 ns)
19 PAL INPUT TO Hi-Z (25 ns)

Figure 4.

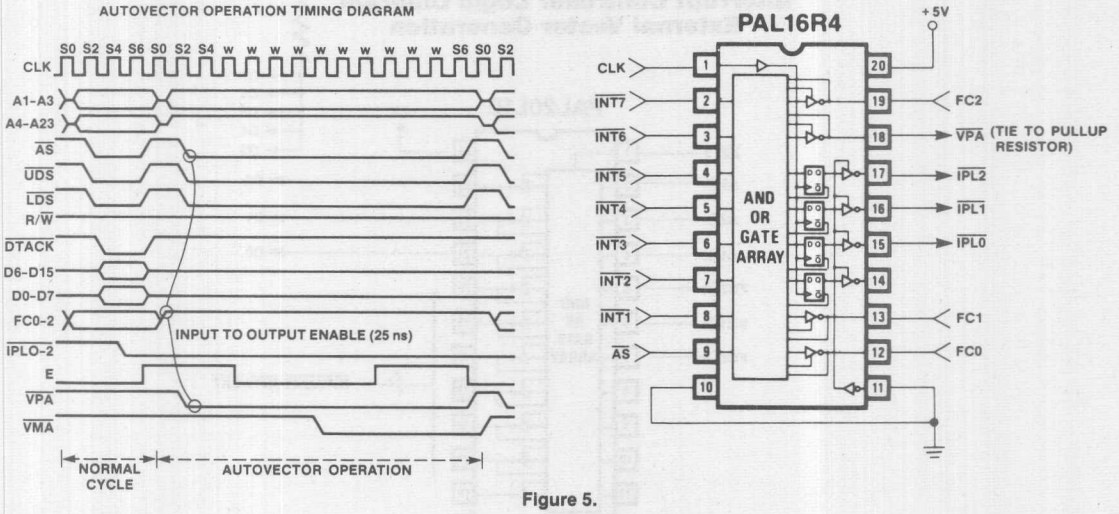


Figure 5.

Conclusions

We have just seen how to implement an interrupt controller circuit using one or two PAL devices. This circuit can operate at the maximum operating frequency of the 68000 which is 12.5 MHz. Since the most critical timing of the circuit is the PAL device unit's input to Hi-Z, we see that we can operate the circuit with a wide spectrum of frequencies and with slower PAL devices. To guarantee operation, timing spec. No. 16 and No. 19 on Figure 4 must add up to assert DTACK 20 ns prior to falling edge of S4. Thus 55 ns + (PAL device input delay to Hi-Z) should be less than. or equal to, 100 ns for a 12.5-MHz clock. We see that this qualifies fast, regular, half-power and quarter-power PAL devices for this circuit.

```
TITLE     INTERRUPT CONTROLLER1
PATTERN   INTA.DAT
REVISION  01
AUTHOR    DANESH TAVANA
COMPANY   MONOLITHIC MEMORIES, INC.
DATE      11/06/87

CHIP  INTA  PAL16R4

CLK   /INT7  /INT6   /INT5  /INT4  /INT3  /INT2  /INT1   /AS  GND
/OC   FC0    FC1     NC     /IPL0  /IPL1  /IPL2  /DTACK  FC2   VCC

EQUATIONS

DTACK = VCC                                  ;ASSERT IF OUTPUT ENABLE
DTACK.TRST = FC2*FC1*FC0*AS

IPL2 :=  INT7                                ;PRIORITY ENCODED BIT
       + /INT7* INT6
       + /INT7*/INT6* INT5
       + /INT7*/INT6*/INT5* INT4

IPL1 :=  INT7                                ;PRIORITY ENCODED BIT
       + /INT7* INT6
       + /INT7*/INT6*/INT5*/INT4* INT3
       + /INT7*/INT6*/INT5*/INT4*/INT3* INT2

IPL0 :=  INT7                                ;PRIORITY ENCODED BIT
       + /INT7* INT6
       + /INT7*/INT6* INT5*/INT4* INT3
       + /INT7*/INT6*/INT5*/INT4*/INT3*/INT2* INT1

; SIMULATION NOT INCLUDED
```

```
TITLE     INTERRUPT CONTROLLER2
PATTERN   INTC.DAT
REVISION  01
AUTHOR    DANESH TAVANA
COMPANY   MONOLITHIC MEMORIES, INC.
DATE      11/06/87

CHIP INTC PAL20L10

/AS     A3     A2     A1     FC2     FC1     FC0      NC     NC NC NC GND
NC   /INTACK1 /INTACK2 /INTACK3 /INTACK4 /INTACK5 /INTACK6 /INTACK7 D0
D1 D2 VCC

EQUATIONS

/D2 = /A3
/D2.TRST = FC2*FC1*FC0*AS

/D1 = /A2
/D1.TRST = FC2*FC1*FC0*AS

/D0 = /A1
/D0.TRST = FC2*FC1*FC0*AS

INTACK7 = A3* A2* A1
INTACK7.TRST = FC2*FC1*FC0*AS

INTACK6 = A3* A2*/A1
INTACK6.TRST = FC2*FC1*FC0*AS

INTACK5 = A3*/A2* A1
INTACK5.TRST = FC2*FC1*FC0*AS

INTACK4 = A3*/A2*/A1
INTACK4.TRST = FC2*FC1*FC0*AS

INTACK3 =/A3* A2* A1
INTACK3.TRST = FC2*FC1*FC0*AS

INTACK2 =/A3* A2*/A1
INTACK2.TRST = FC2*FC1*FC0*AS

INTACK1 =/A3*/A2* A1
INTACK1.TRST = FC2*FC1*FC0*AS

; SIMULATION NOT INCLUDED
```

Memory Control

Memories are an integral component of any computer system. PROMs (Programmable Read Only Memory), SRAMs (Static Random Access Memory), and DRAMs (Dynamic Random Access Memory) are used most often to store programs and data. PROMs are used to store programs which can only be read by the computer. The SRAMs and DRAMs are used for reading and writing data and programs. Figure 1 shows a system processor interfaced to PROMs, SRAMs and DRAMs.

DRAMs are cheaper bulk memory, and are used instead of SRAMs in many systems. DRAMs are available in speeds ranging from 70 ns to 250 ns access times, enough to keep up with the new generation processor speeds. This performance is also sufficient to match the usually high speed SRAMs in many existing and new designs. DRAM densities have also progressed from 64K-bit in 1983 to 1M-bit in 1986, and 4M-bit and 16M-bit DRAMs are being developed. DRAMs are also available in space efficient SIP, SIMM and ZIP (Zig-zag Inline Package) packages with modules that can carry up to eight or nine DRAM devices. In this discussion we will focus on the design of PROM, SRAM and DRAM control circuitry using PLDs.

Large memory requirements result in the use of many DRAMs which cannot be conveniently located on the same board with the CPU. DRAM devices are placed on separate boards and connected to the processor board via a bus. Off-board, DRAM devices mandate the use of specialized outputs to drive the address and control traces of the DRAM array. In systems with many off-board DRAMs the soft error rate is high and sometimes calls for the use of error detection and correction circuits. A detailed discussion of EDC (Error Detection and Correction) circuit design with PLDs is also included here.

The proximity of the DRAMs to the processing units depends on the hierarchy of the DRAM memory in the system. This hierarchy depends heavily on the speed of the DRAM system versus the processor speed, and on the required addressing space. When the processor speed can be accommodated by the DRAM devices, it is efficient to place a DRAM subsystem as close to the processor as possible and to use it as the immediate storage space. When DRAMs are not fast enough for the processor, they must be used as remote, main-storage, while static-RAM-based caches are used in the processor's immediate proximity. A review of such cache systems design using PLDs is also included here.

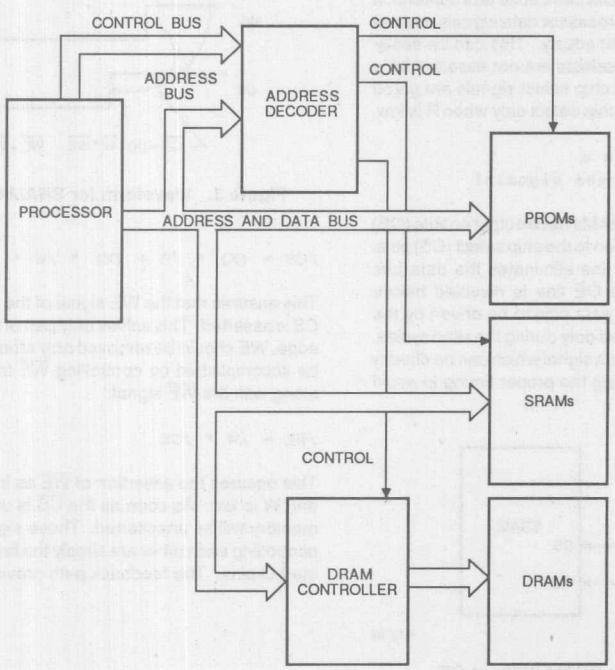


Figure 1. Processor Interface to PROMs, SRAMs and DRAMs

PROM and SRAM Control

The control of static RAM or PROM devices is very simple. Both of these devices provide a chip select input signal which activates the devices to read or write data. The chip select control of these devices is enabled by the decoded processor address signals. The address bits selected depend upon the range of addresses covered by each SRAM or PROM. More detail on address decoders can be found on page 2-35.

To optimize the number of product terms used, the address range is generally selected so that the boundary addresses are power-of-two numbers. From page 2-53, for an address decoder we have the following equations:

For an n-bit address X in the range from B+1 to A–1 (A>X>B)

```
A= 0010000—00 (a power-of-two number)
   n p      1
B= 0000011—11 (a power-of-two number minus one)
   n   q    1

CS = /Xn * /Xn-1 * - - - * /Xp *
       (Xp-1 + Xp-2 + - - - + Xq)
```

Typical SRAMs also include a read/write input signal which selects between a read and a write operation. This signal is usually directly connected to the processor read/write signal. PROMs are "read only", and thus do not require this signal.

PROM chip select signals should only be asserted for read cycles. Attempting a write to a PROM can cause bus contention problems where the memory and processor data signals attempt to drive the same bus lines simultaneously. This can be easily avoided by ensuring that the chip selects are not asserted for a processor write cycle. The PROM chip select signals are gated with the read signal (R) to insure a chip select only when R is low.

```
/CS_PROM = (X) * /R ; (X is a
       combination of address signals)
```

Most of the "second generation" SRAMs have output enable (OE) and write enable (WE) pins in addition to the chip select (CS) pins. A separate and independent OE line eliminates the data bus contention problem entirely. The OE line is disabled before writing to the SRAM, allowing the data pins to be driven by the external bus. The OE line is enabled only during the read cycles. Most microprocessors provide a data signal which can be directly connected to the OE lines, providing the proper timing to avoid bus contention.

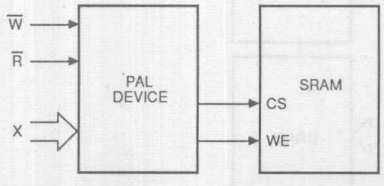


Figure 2. Controlling an SRAM Without OE

Some memories so not provide a separate OE pin (Figure 2). Generating CS for these memories can cause serious bus contention problems. Often this problem is solved by gating the CS with read (R) or write (W) signals as follows:

```
/CS = (X) * /R
    + (X) * /W

/WE = /W
```

This approach allows usually sufficient time for other buffers on the data bus to be disabled. This works well for the read cycles in which the memory is expected to be the only bus driver. However, for the write cycles where the processor is usually the data bus driver, this does not quite solve the problem completely.

In the above equation, the CS is generated either from the read or the write signal. If the CS line is generated before the WE is asserted low, the memory will drive the data lines, which are already being driven by the processor, causing bus contention. There is still no way to ensure that CS is generated only after the WE signal to the memory is asserted. Some delay can be added by controlling the CS line from the WE signal, by gating the $\overline{W}$ signal with the WE signal (for CS) (Figure 3) as follows:

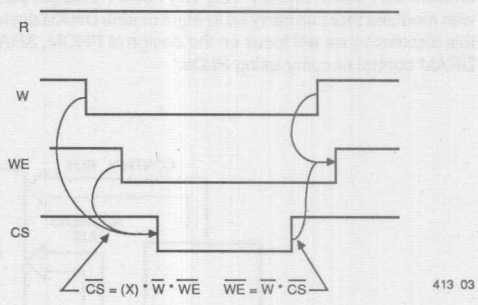


Figure 3. Waveform for SRAM Control Without OE

```
/CS = (X) * /R + (X) * /W * /WE
```

This ensures that the WE signal of the memory is low before the CS is asserted. This solves only part of the problem. On the other edge, WE should be removed only after CS is removed. This can be accomplished by controlling $\overline{WE}$ from the $\overline{CS}$ by gating $\overline{CS}$ along with the $\overline{WE}$ signal:

```
/WE = /W * /CS
```

This ensures the assertion of $\overline{WE}$ as long as the $\overline{CS}$ is asserted and W is low. As soon as the $\overline{CS}$ is unasserted, the $\overline{WE}$ of the memory will be unasserted. These signals $\overline{CS}$ and $\overline{WE}$ used for controlling each other are simply the feedback from the appropriate I/O pins. The feedback path provides the extra delay.

Access Time Considerations

PROMs and SRAMs with data access time which match the processor speeds are commercially available. Sometimes, however, there is a speed mismatch. Systems with different types of memories encounter this problem the most. For such systems, different memories generate their own data available (DAV) and write complete signals and inform the processor of cycle completion. This allows processors to prolong their cycles and add wait states, to match the memory speeds. These signals are generated by using PAL devices which can generate timing delays. Registered PAL devices are ideal candidates for such timers. The application note on page 2-184 illustrates one such memory design where a PAL device generates the memory control timing signals. This design also implements the memory-to-processor interface handshake logic.

DRAM Systems

In any system that uses dynamic RAMs, special circuitry in the address and control lines must be introduced to drive the memory chips and link them to the CPU (see Figure 4). A circuit known as a DRAM controller performs several functions:

- Multiplexes the row and column addresses
- Provides access timing
- Provides refresh timing and addresses
- Arbitrates between access and refresh
- Drives the DRAMs' capacitive load inputs

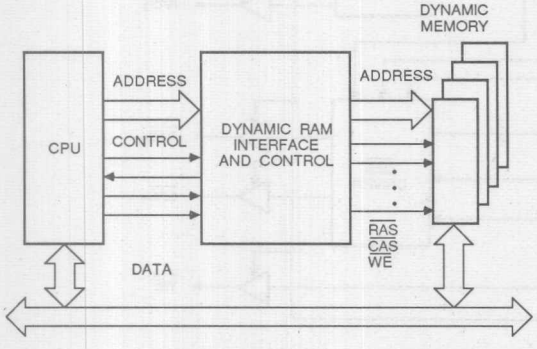


Figure 4. Functions of a DRAM Controller

Multiplexing processor addresses into row and column addresses is required because the DRAM architecture divides the addresses into rows and columns. This reduces the number of pins required for addressing the DRAMs and consequently the package size. The external DRAM controller multiplexes the addresses and also generates the RAS (Row Address Strobe) and CAS (Column Address Strobe) signals to the DRAMs to distinguish between these two addresses. Every DRAM access, for reading or writing data, requires a predetermined timing sequence of RAS, CAS and WE (Write Enable) signals. A DRAM controller's main task is to generate these read and write cycle timings. In addition to the regular read and write cycles, many DRAMs are available which provide higher data bandwidth read and write cycle. Such nibble mode, ripple mode or page mode cycles require different sequences of RAS and CAS signals. Depending upon the data requirements, these schemes are used for high speed data throughput in various systems.

DRAMs also require intermittent refresh cycles to charge the capacitors for every row of data. This refresh is provided by a specific sequence of RAS and CAS signals. There are three different refresh cycles, one of which is selected according to the system requirements. These are:

- Regular RAS only refresh cycle
- CAS before RAS refresh cycle
- Hidden refresh cycle.

The regular RAS-only refresh cycle requires an external refresh address generator which sequences through all the row addresses of a DRAM. The CAS before RAS refresh cycle is only provided by certain DRAMs. These DRAMs are usually costlier, since they require internal circuitry for row address generation. The hidden refresh cycle DRAMs retain the last read/write cycle data at the outputs during the refresh cycles, which is helpful in certain designs.

In addition to the type of refresh cycle, the frequency of refresh cycles also varies in different systems. Some systems provide intermittent refresh when refresh cycles are executed at fixed time intervals. The DRAM accesses are sometimes delayed pending an intermittent refresh cycle. With burst mode refresh, a number of refresh cycles are completed simultaneously, allowing the processor an uninterrupted access for read and write cycles later. Time critical systems usually find burst mode more useful.

DRAM Controllers and PLDs

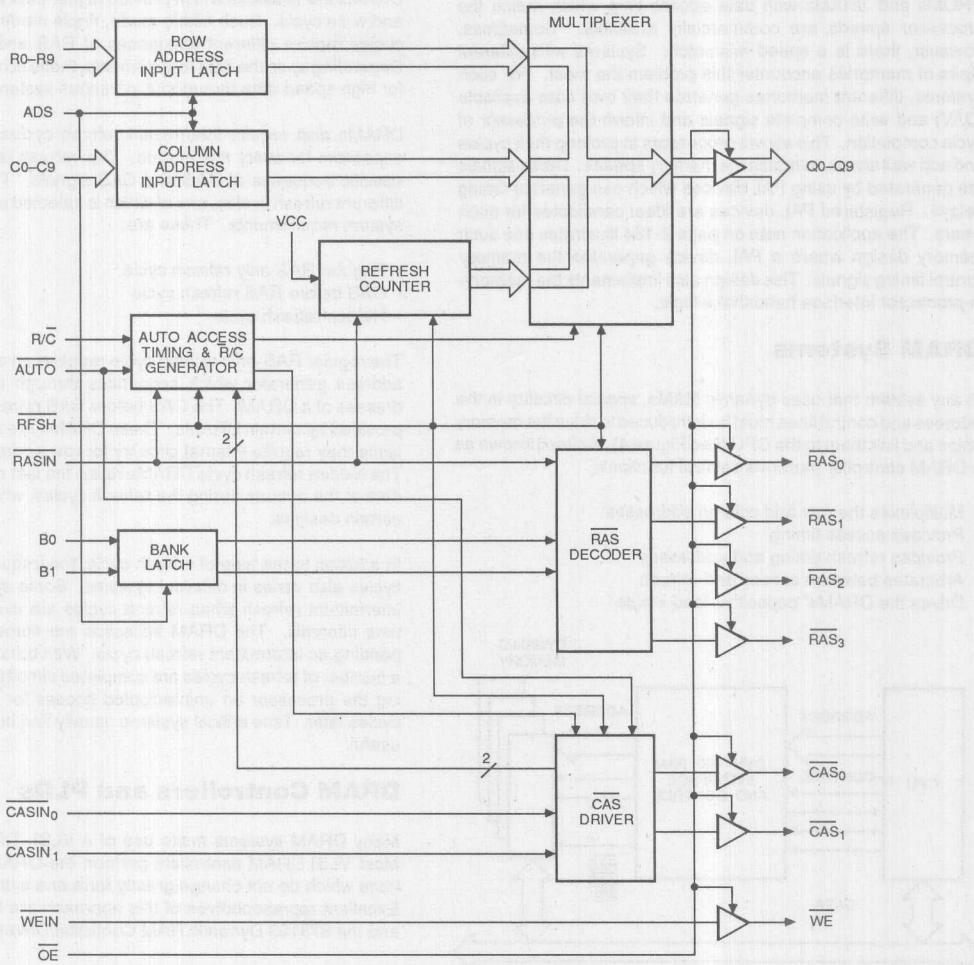
Many DRAM systems make use of a VLSI DRAM controller. Most VLSI DRAM controllers perform the DRAM control functions which do not change greatly form one system to another. Excellent representatives of this approach are the 2964, 2968 and the 673103 Dynamic RAM Controller/Drivers (Figure 5).

Most VLSI DRAM controllers do not support all of the various types of read/write cycles, refresh cycles and refresh timings. Custom DRAM controllers are required for the following reasons:

- Higher throughput bandwidth
- Different memory access times
- Different refresh cycles and mechanisms
- Processor related requirements.

Depending upon the specific system requirements, DRAM controllers can be made from PLDs which support the specific cycles required by a design.

Fast registered PLDs are used as DRAM controllers in many designs. Functions of a DRAM controller are usually split into a number of PLDs. The application note on page 2-187 shows a



413 05

Figure 5. A VLSI DRAM Controller

DRAM controller for a 68020 microprocessor which implements a custom data sizing and alignment function required by the microprocessor.

VLSI DRAM controllers use PAL devices for a variety of functions. These functions depend upon the capabilities of individual VLSI controllers. Three detailed application notes are included in the subsequent sections which illustrate the interface of these VLSI DRAM controllers to the various microprocessors. The application note on page 2-202 shows an AMD 8088 microprocessor interfaced to a 2968 DRAM controller using a PAL16R4. The application note on page 2-210 shows a 68000 microprocessor interface to the 2968. The interface between the DRAM controller and the CPU accommodates the specific access protocols and speeds of the CPU as well as the access and refresh timing of the dynamic RAMs. This interface between the controller and the rest of the system is best implemented using fast programmable logic devices such as the PAL16R8D family devices.

Error Detection and Correction

Compared to PROMs and SRAMs, the physical dimensions, signal levels, and the stored charge of the DRAM cells are greatly reduced to allow higher density. As the stored charge in the DRAM cells decreases, the device is more susceptible to soft errors caused by alpha particles as well as by environmental noise.

Soft errors are temporary, random, and they can be corrected by rewriting into the erroneous cell. Therefore, if the system has the ability to locate the bit in error, the soft error can be corrected. One method to locate the bit in error is to append a fixed number of check-bits to the data word and store these check-bits along with data. This increases the data width of the DRAM system. Upon reading, both the data and check-bits are read into the EDAC. The check-bits are regenerated from the read data, and these newly generated check-bits are compared with the read check-bits. The comparison produces the syndrome bits. The syndrome bits constitute a binary number that points to the bit in error.

This encoding scheme of generating the check-bits and the syndrome bits is called a Hamming code, after its inventor. The modified Hamming code has one more check-bit than the original Hamming code, but it can detect all double-bit errors as well as detect and correct all single-bit errors. In addition it can detect a substantial number of multiple-bit errors. The modified Hamming code has proved particularly useful for its relatively low overhead in today's wide-data-word systems and its capability to detect and correct single-bit errors and detect all double-bit errors.

A number of XOR logic functions are required for generating Hamming codes. PAL devices which provide XOR gates are ideal candidates for such designs. Details of Hamming codes and a sample eight-bit error detection and correction application note which uses the PAL16X4 is included on page 2-229.

Cache Memory Systems

Fast access time SRAM caches are used for very high-speed processors. These allow high system performance while retaining the low cost of DRAM bulk memory. Based upon how the SRAM cache addresses map to the DRAM addresses, three major cache schemes are used.

For fully associative caches, data in a DRAM location may be in any cache location. For direct mapped caches, data may be in one of the many locations determined by higher order address bits. For set associative caches, data may be in one particular location of every set. Depending upon the performance required and other such system constraints such as cost, board space etc. one of the three schemes is used. Details of cache system design are included in the application note on page 2-239.

A major task in cache memory systems is replacement, which requires transferring data between the cache and the DRAMs. This is done in an effort to keep the cache filled with only those blocks of memory which are in use by a program. The replacement task requires a controller with intelligence, which can read cache data and write to the DRAMs and vice versa. This requires a state-machine-based controller. Since most cache designs are custom designs, due to various system constraints, limited VLSI solutions are available. State-machine-based controllers such as the Am29PL141 and the PMS14R21 can perform these functions very effectively.

Other uses of PLDs in cache memory systems include general glue logic which is required for controlling SRAM caches or tag buffers. Such PLDs are usually very high-speed combinatorial PLDs. Monolithic Memories also provides a number of ECL PLDs which can be used in ECL cache memory systems. These ECL PAL devices allow 6 ns access times.

Memory Handshake Logic

A typical control logic problem is the memory-to-processor handshake on memory transfer used in many computer architectures. The processor makes a transfer request by activating a request line (REQ) and specifies a read or write operation on a Read/Write line (R/W).

During a read operation, the processor waits for a Data Available signal at which time the data bus is sampled and the request line lowered, thus completing the cycle. During a write operation, the processor places data on the bus and waits for a Write Complete signal after the write cycle is finished. Upon write complete, the request line is lowered, hence completing the cycle. Figure 2 shows the state assignments and the appropriate outputs. The state diagram is shown in Figure 1. Also the handshaking operation is illustrated in the timing diagram of Figure 3.

The memory-board logic to implement this function may be designed with gates and edge-triggered flip-flops as shown in Figure 4. This particular design would require about five SSI/MSI packages, but the same design can be implemented by a single PAL16RP6. The PAL device design specification using state equations is shown on page 2-186.

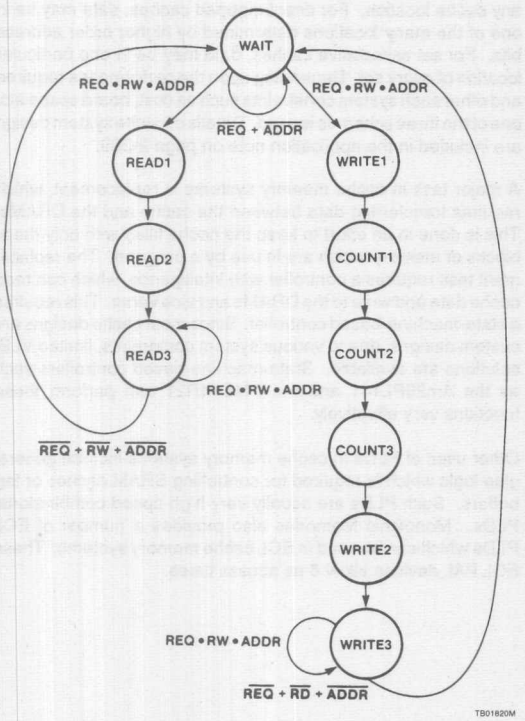


Figure 1. State Diagram—Memory Handshake Logic

STATE	DOUT	DA	WE	WC	C0	C1
WAIT	0	0	0	0	0	0
READ1	1	0	0	0	0	0
READ2	1	1	0	0	0	0
READ3	0	0	0	0	0	0
COUNT1	0	0	1	0	1	0
COUNT2	0	0	1	0	0	1
COUNT3	0	0	1	0	1	1
WRITE1	0	0	1	0	0	0
WRITE2	0	0	1	1	0	0
WRITE3	0	0	0	1	0	0

Figure 2.

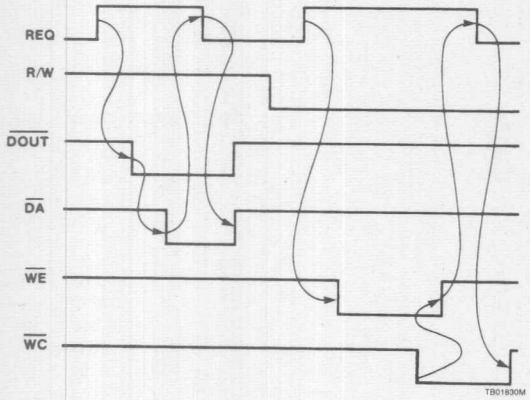


Figure 3. Memory Handshake Timing

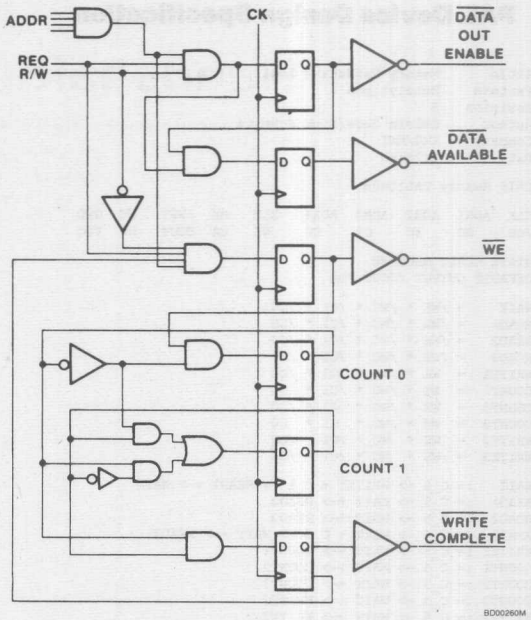
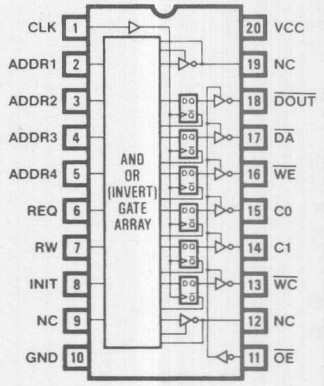


Figure 4. Memory Handshake Logic

PAL Device Design Specification

```
Title       Memory Handshake Logic
Pattern     Memory1.pds
Revision    B
Author      Kelvin Chow/Nick Schmitz
Company     AMD/MMI
Date        2/28/85

CHIP Memory PAL16RP6

CLK  ADR1  ADR2  ADR3  ADR4   REQ   RW  INIT   NC  GND
/OE    NC    WC    C1    C0    WE   DA  DOUT   NC  VCC

STATE MEALY_MACHINE
DEFAULT_OUTPUT /DOUT /DA

WAIT    = /WE * /WC * /C1 * /C0
READ1   = /WE * /WC * /C1 * /C0
READ2   = /WE * /WC * /C1 * /C0
READ3   = /WE * /WC * /C1 * /C0
WRITE1  =  WE * /WC * /C1 * /C0
COUNT1  =  WE * /WC * /C1 *  C0
COUNT2  =  WE * /WC *  C1 * /C0
COUNT3  =  WE * /WC *  C1 *  C0
WRITE2  =  WE *  WC * /C1 * /C0
WRITE3  = /WE *  WC * /C1 * /C0

WAIT   := C_2 -> WRITE1 + C_1 -> READ1 +-> WAIT
READ1  := C_5 -> WAIT +-> READ2
READ2  := C_5 -> WAIT +-> READ3
READ3  := C_4 -> WAIT + C_5 -> WAIT +-> READ3
WRITE1 := C_5 -> WAIT +-> COUNT1
COUNT1 := C_5 -> WAIT +-> COUNT2
COUNT2 := C_5 -> WAIT +-> COUNT3
COUNT3 := C_5 -> WAIT +-> WRITE2
WRITE2 := C_5 -> WAIT +-> WRITE3
WRITE3 := C_1 -> WRITE3 + C_4 -> WAIT + C_5 -> WAIT

READ1.outf  :=  DOUT * /DA      ; 2
READ2.outf  :=  DOUT *  DA      ; 3

CONDITIONS

C_1 = REQ *  RW * ADR1 * ADR2 * ADR3 * ADR4 * /INIT
C_2 = REQ * /RW * ADR1 * ADR2 * ADR3 * ADR4 * /INIT
C_3 = (/REQ + /ADR1 + /ADR2 + /ADR3 + /ADR4) * /INIT
C_4 = (/REQ + /ADR1 + /ADR2 + /ADR3 + /ADR4 + /RW) * /INIT
C_5 = INIT

SIMULATION
TRACE_ON CLK REQ RW DOUT  DA  WE  WC C1 C0

SETF INIT /REQ OE RW ADR1 ADR2 ADR3 ADR4
CLOCKF CLK      CLOCKF CLK
CLOCKF CLK      CHECK wait /DOUT

SETF REQ /INIT
CLOCKF CLK      CHECK read1
CLOCKF CLK      CHECK read2
CLOCKF CLK      CHECK read3
CLOCKF CLK      CHECK read3

SETF INIT
CLOCKF CLK      CHECK wait
CLOCKF CLK      CHECK wait
CHECK /DOUT /DA

SETF /INIT REQ /RW
CLOCKF CLK      CHECK write1
CLOCKF CLK      CHECK count1
CLOCKF CLK      CHECK count2
CLOCKF CLK      CHECK count3
CLOCKF CLK      CHECK write2
CLOCKF CLK      CHECK write3

TRACE_OFF
```

Logic Symbol



Customize a DRAM Controller Using Advanced PAL Devices

Ever increasingly dense DRAMs and their interface to the emerging sophisticated 32-bit microprocessors provide an increasing number of system options to the designer. In order to utilize these advanced DRAM features and to satisfy the individual system design requirements, custom DRAM controllers are required. PAL device sequencers, such as the PAL23S8, provide effective control sequencing power, flexibility, customizability and ease of use. They offer a very desirable alternative to the dedicated functionality of VLSI DRAM controllers.

The following megabit DRAM controller design based on the PAL23S8 sequencer is illustrative of a design which is easily customized to a system requirement. It supports features such as data bus sizing and data alignment which are required for the emerging 32-bit microprocessors, but which are not yet easily and adequately supported by the VLSI controllers.

A DRAM controller needs to support at least the following functions:

- Flexible Refresh Generation
- Flexible Arbitration
- Processor Handshake
- Processor Cycle Timing and Control
- Refresh Cycle Timing and Control

Flexible Refresh Generation Mechanisms

For refresh generation, different system designs may require either burst mode, intermittent mode or both kinds of refresh. In burst mode, refresh requests occur once very 4 milliseconds to meet the dynamic RAM's speed. For a megabit RAM, which needs 512 cycles for a complete refresh, all 512 cycles will be performed consecutively. The major disadvantage of the Burst refresh approach is that memory is "out of service" for a long time. The intermittent refresh scheme minimizes the out of service time by stealing single cycles between processor cycles. This cycle stealing is performed by a pre-defined arbitration mechanism for prioritization of different cycle requests. This design incorporates the arbitration in a manner which allows both intermittent and burst mode refreshes.

Flexible Arbitration Scheme

The arbitration in most designs involves prioritizing memory accesses for: read cycles, write cycles, read-modify-write cycles, and refresh cycles.

The DRAM controller must allow both asynchronous memory and refresh cycle requests. It should decide whether a memory cycle is an access cycle or a refresh cycle. When refresh and processor accesses are requested simultaneously, the refresh is given priority. If a refresh cycle is in progress and a memory cycle request is made, then the memory access is inhibited until the refresh is complete. The memory request in this case is kept pending by asserting a processor wait signal.

For a typical intermittent refresh scheme, the cycle requests are derived from the oscillator, which operates asynchronously with respect to the system clock. The arbiter grants a refresh request when no other memory cycle is either occurring or pending. If a memory cycle is in progress, the arbiter inhibits the refresh cycle until the current cycle is completed. However, it needs to remember that a refresh request was made and grant that request on completion of the processor cycle. The PAL23S8's buried registers are very useful in performing such arbitration functions, even between asynchronous events. These registers can also be used for remembering refresh cycle requests and initiating actions later for execution of the refresh cycles.

However, some designs may require additional functions, such as prioritizing different processor requests. Some may require a specific kind of handshake mechanism between the processor and the DRAM controller. All such functions can be performed in the PAL sequencer by using more of chip resources such as I/O pins, buried registers and product terms. On the other hand some systems may not require functions such as read-modify-write cycles, in which case the PAL sequencer resources can be saved and used for other system functions.

Flexible Handshake Protocol

The PAL23S8 sequencer can provide a customizable interface to the system processor without requiring additional parts. The logic required for interface protocols for the APX family or the 68000 family microprocessors can be very easily built using the internal resources of the PAL23S8. The design example below is for 68020 processor interface signals.

Processor Cycle Execution

Different DRAMs can have different processor cycles. Some just support the basic read/write cycles, whereas others support extended functions such as page mode, ripple mode or static column mode cycles. Some DRAMs also support additional functions for extended applications such as the transfer cycle in VRAMs (DRAMs optimized for video applications). PAL devices allow user-definable functionality to be implemented in the design, which optimizes resource utilization and improves system performance. Another advantage offered by such designs is the customizability of the design to the DRAM timing requirements, which can vary between -12, -15, and -20 parts significantly. All these timing functions can be easily implemented and modified using buried state registers.

Our design implements the basic read, write, and read-modify-write cycles required in most designs. The timing required has

been calculated for a -12 memory part. The processor cycle execution involves the following functions:

- Address multiplexing
- Timing and control signal generation

Am29841 ten-bit buffers are used for address multiplexing. The multiplexing of the row or column address is under the control of the DRAM controller.

Generation of the timing and control signals RAS, CAS, WE and DACK is also the core function of the controller. The controller multiplexes the address and produces RAS, CAS, WE in a sequence that is understandable by the DRAMs. It also generates a handshake signal "DACK" for the CPU, to indicate whether or not the memory is ready to be accessed.

Refresh Cycle Execution

DRAMs support different types of refresh cycles. Some DRAMs support CAS before RAS refresh. These DRAMs have the refresh address counter embedded along with the memory, thus eliminating the need for an external counter. Some designs may require "hidden refresh", or the standard RAS only refresh. Depending upon the system requirements, the designer can optimize the design and resource utilization. For this example, the PAL device has been designed to execute a RAS only refresh. However, it can be easily modified to support CAS-before-RAS refresh, or hidden refresh also, if needed.

Megabit DRAM Controller with Data Alignment and Dynamic Bus Sizing

In many cases the currently available DRAM controllers do not support the esoteric requirements of different system designs. The designer is left to make a custom DRAM controller. The PAL23S8, with its embedded extra buried state registers, provides an excellent tool for designing such a flexible, fast DRAM controller. It also offers the right speed for interfacing to the emerging 8 and 16 MHz processors, allowing superior synchronous designs with few or no wait states.

Some of the emerging 32-bit microprocessors, such as the 68020 and the 80386, support the concept of dynamic bus sizing and mis-aligned data transfers. Presently, there is no DRAM controller which utilizes the advantages offered by these two features effectively.

Dynamic Data Bus Sizing

Dynamic bus sizing allows the processor to handle variable width data buses (8, 16 or 32 bits) on a cycle-by-cycle basis. Dynamic bus sizing makes the size of the R/W port or channel transparent to the software designers.

The 68020 allows operand transfers to or from 8, 16, and 32-bit ports by dynamically determining the port size during each bus cycle. Similarly, 80386 allows operand transfers to or from 16 and 32-bit ports by determining the port size during each bus cycle. During an operand transfer, the slave device signals its port size and transfer status (complete or not) to the processor using the DSACK inputs for the 68020, and the Ready & BS16 signals for the 80386.

DSACK1	DSACK0	
H	H	Insert Wait States
H	L	Complete Cycle Data bus port size 8 bits
L	H	Complete Cycle Data bus port size 16 bits
L	L	Complete Cycle Data bus port size 32 bits

The design implemented shows a 68020 interface, but can be easily modified to accommodate the 80386. The 68020 always attempts to transfer the maximum possible amount of data (32 bits) in a single cycle. If the port responds with a size smaller than 32 bits, the processor repeats the cycles with the remaining bytes of data. For each cycle, the 68020 informs the slave of the number of bytes it is attempting to transfer by two signals SIZ0, SIZ1, as follows:

SIZ0	SIZ1	SIZE
0	1	Byte
1	0	Word
1	1	3 Bytes
0	0	Long Word

Due to various considerations, memories with different data width can exist on the same system. For example, it is often preferable to have smaller external bus sizes, while retaining a 32-bit internal memory size on a board. Such instances are common when using DMA in microprocessor based systems. Varied memory widths are also required in graphics and image processing systems. PAL devices allow customizable dynamic memory data bus sizing based on various system requirements—such as address range (address signals) or jumper selectable static control for increasing memory width.

Mis-Aligned Transfers

Dynamic sizing is used in conjunction with address bits A0 and A1 to also support mis-aligned data transfers; for example, a word write to a long word data bus size with an address offset of 1 byte (A0/A1=01). The 68020 and the 80386 contain internal bus multiplexers to route the data to the correct data lines. Both of these also generate extra cycles necessary to complete a mis-aligned transfer. However, external circuitry is still needed to generate the correct write enables and chip selects to support both these functions. PAL devices, however, can be customized to support both these functions easily.

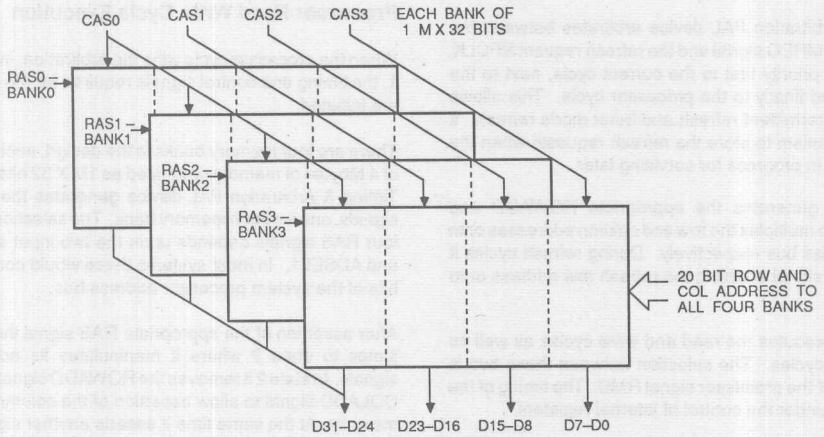
A1	A0	OFFSET
0	0	0 Bytes
0	1	1 Byte
1	0	2 Bytes
1	1	3 Bytes

By decoding A0 and A1, the two transfer data size signals SIZE0, SIZE1, and the size of the port being addressed, PAL devices can easily support dynamic bus sizing and generate the appropriate chip select signals for the four banks of memory.

The memory organization selected (Figure 1) allows access to individual data bytes under the control of the CAS signal. The four banks of memory are addressed by the four RAS signals.

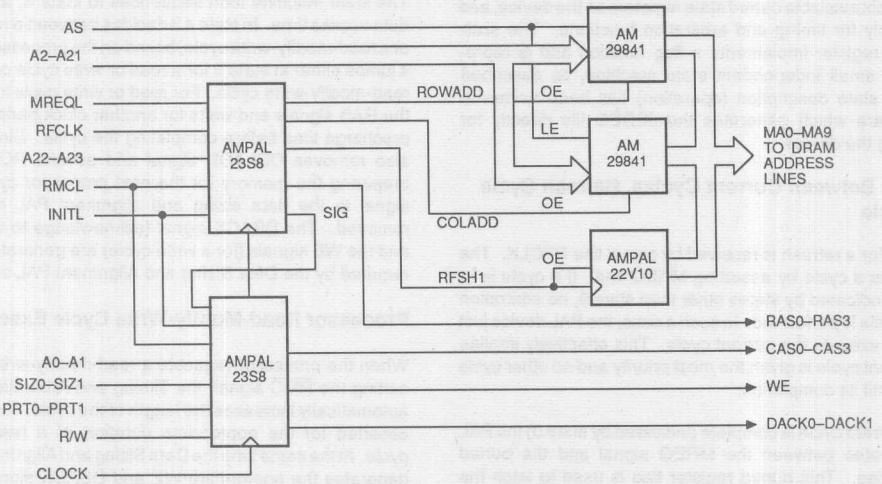
The DRAM Controller

The DRAM controller consists of two PAL23S8s: the first is called the Timing and Arbitration PAL device, and the second device is called the Data Sizing and Alignment PAL device (Figure 2).



441 02

Figure 1. Memory Organization



441 02

Figure 2. A Megabit DRAM Controller

The Timing and Arbitration Controller

The timing and arbitration PAL device performs the following functions:

- Arbitration between current cycle, Refresh cycle & CPU cycle
- Read and Write Cycle Execution
- Read-Modify-Write Cycle Execution
- Refresh Cycle Execution
- Interface Signaling

The timing and arbitration PAL device arbitrates between the processor request MREQ signal and the refresh request RFCLK, giving the highest priority first to the current cycle, next to the refresh request and finally to the processor cycle. This allows support for both intermittent refresh and burst mode refresh. It also uses a mechanism to store the refresh requests when the processor cycle is in progress for servicing later.

This device also generates the appropriate ROWADD and COLADD signals to multiplex the row and column addresses onto the memory address bus respectively. During refresh cycles it asserts the RFSH signal enabling the refresh row address onto the DRAMs.

This PAL device executes the read and write cycles as well as read-modify-write cycles. The selection between these two is under the control of the processor signal RMC. The timing of the different cycles is under the control of internal registers.

The timing requirements of the different cycles are shown in Figure 3. The operation of the state machine to perform all these functions is also shown in Figure 4. This state diagram has been derived directly from the timing diagrams and the arbitration requirements of the design. This state machine is implemented by five of the six available buried state registers on the device, and is used mainly for timing and arbitration functions. The sixth buried state register implements a flag function and is represented as a small independent state machine, as described below. The state description (operation) has been written in CUPL software which generates the JEDEC file directly for programming the device.

Arbitration Between Current Cycles, Refresh Cycle & CPU Cycle

The request for a refresh is received by signal line RFCLK. The CPU requests a cycle by asserting MREQ line. If a cycle is in progress as indicated by states other than state 0, no arbitration for a fresh cycle is performed. In such a case, the PAL device just continues to execute the current cycle. This effectively implies that the current cycle is given the most priority and no other cycle is initiated until its completion.

When the current cycle is complete (indicated by state 0) the PAL device arbitrates between the MREQ signal and the buried register B5 flag. This buried register flag is used to latch the request for refresh (RFCLK) which is usually only one clock cycle duration. The operation of this flag is controlled by a second small state machine (two states only) in the same device. When RFCLK is high this state machine toggles to state 1 (B5=1) and stays in that state till the refresh cycle is executed (indicated by RFSH signal low), and then it toggles back to state 0 (B5=0) preparing itself for the next refresh request RFCLK.

For the first state machine, the arbitration begins on state 0. If B5 is high, a refresh cycle is given priority and the state machine jumps to state 16 to execute a refresh cycle. If on the other hand there is no refresh request pending (indicated by the B5 flag being equal to 0) and MREQ is asserted, a processor cycle is started by jumping to state 1. If none of these conditions exists the state machine stays in state 0, polling for any of these events to occur. Such a scheme can support both burst mode and the intermittent refresh.

Processor Read Write Cycle Execution

When the processor cycle wins the arbitration, indicated by state 1, the timing and control signals required for the processor cycle are initiated.

There are four memory banks in the design, each with a capacity of 4 Mbytes of memory organized as 1M X 32 bits (Figure 1). The Timing & Arbitration PAL device generates the 4 RAS0-RAS3 signals, one for each memory bank. The selection of one of these four RAS signals depends upon the two input signals ADSEL0 and ADSEL1. In most systems these would constitute the high bits of the system processor address bus.

After assertion of the appropriate RAS signal the state machine jumps to state 2 where it manipulates its address handling signals. In state 2 it removes the ROWADD signal and asserts the COLADD signal to allow assertion of the column address to the memory. At the same time it asserts another signal, SIG, which is used to inform the Data Sizing and Alignment PAL device of a processor cycle request, and allows the Data Sizing and Alignment PAL device to generate the appropriate CAS signal to the memory.

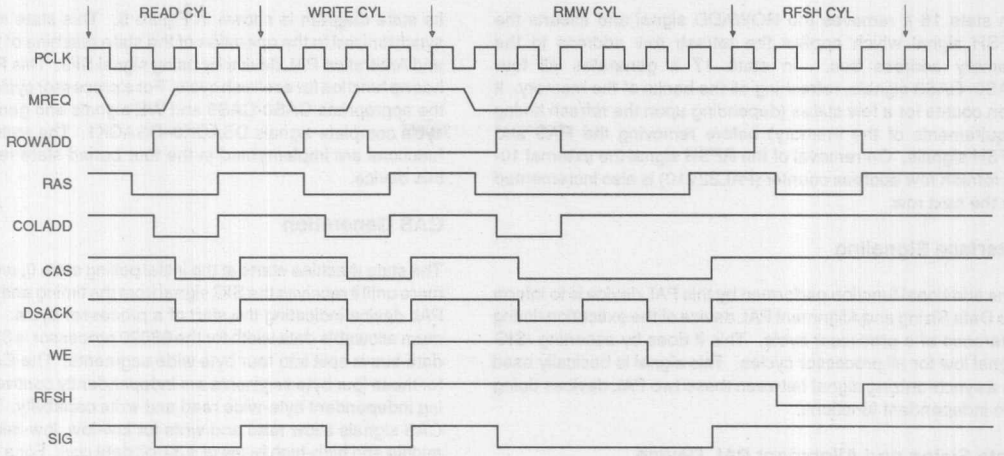
The state machine then sequences to state 4, allowing for the data access time. In state 4 it decides between a read write cycle or a read-modify-write cycle, based on the processor signal RMC; it jumps either to state 5 for a read or write cycle or to state 8 (for read-modify-write cycle. For read or write cycle it then removes the RAS signals and waits for another clock period for the RAS precharge time before completing the cycle. Simultaneously it also removes COLADD signal and asserts ROWADD signal preparing the memory for the next processor cycle. The SIG signal to the data sizing and alignment PAL device is also removed. The DSACK signal (acknowledge to the processor) and the WE signals (for a write cycle) are generated if and when required by the Data Sizing and Alignment PAL device.

Processor Read-Modify-Write Cycle Execution

When the processor requests a read-modify-write cycle by asserting the $\overline{\text{RMC}}$ signal, the Timing and Arbitration PAL device automatically increases the length of the cycle. It keeps the RAS asserted for the appropriate duration of a read-modify-write cycle. At the same time the Data Sizing and Alignment PAL device generates the appropriate WE and DSACK signals. After this extended RAS assertion the Timing and Arbitration PAL device completes the cycle, similar to a read or write cycle.

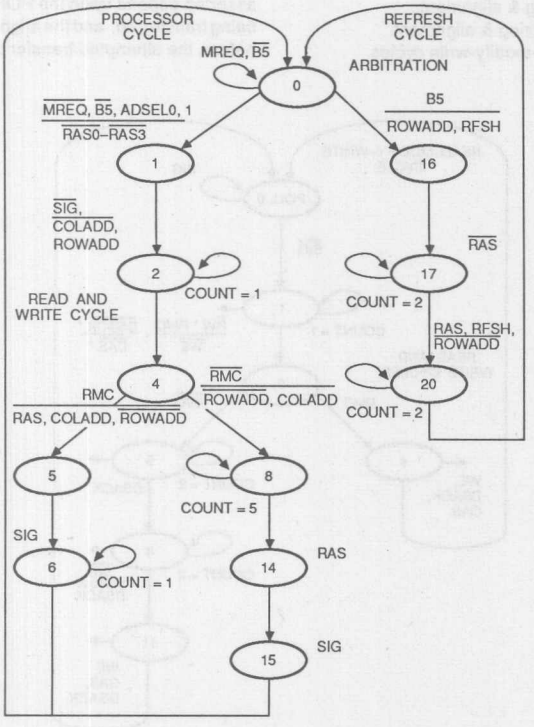
Refresh Cycle Execution

When the arbitration is won by the refresh cycle, the timing and arbitration control PAL device executes a RAS-only refresh cycle.



441 03

Figure 3. The Timing Diagram



441 04

NOTE: WHERE SIGNAL DESIGNATIONS APPEAR BOTH ABOVE AND BELOW A LINE, THE UPPER INDICATES INPUTS AND THE LOWER, OUTPUTS.

Figure 4. State Diagram for Timing and Arbitration PAL Device

On state 16 it removes the ROWADD signal and asserts the RFSH signal which applies the refresh row address to the memory address bus. In state 17 it generates all four RAS0–RAS3 signals, refreshing all the banks of the memory. It then counts for a few states (depending upon the refresh timing requirements of the memory) before removing the RAS and RFSH signals. On removal of the RFSH signal the external 10-bit refresh row address counter (PAL22V10) is also incremented for the next row.

Interface Signaling

One additional function performed by this PAL device is to inform the Data Sizing and Alignment PAL device of the execution timing reference of a processor cycle. This it does by asserting /SIG signal low for all processor cycles. This signal is basically used as a synchronizing signal between these two PAL devices doing two independent functions.

Data Sizing and Alignment PAL Device

The Data Sizing and Alignment PAL device performs the following functions:

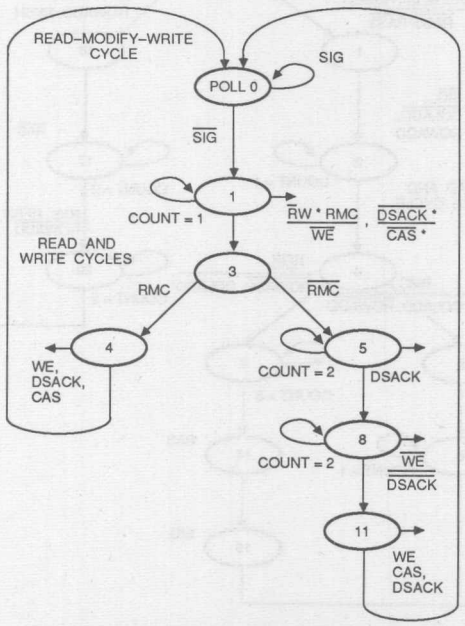
1. CAS generation for dynamic sizing & alignment.
2. DSACK generation for dynamic sizing & alignment.
3. WE generation for write and read-modify-write cycles.

Its state diagram is shown in Figure 5. This state machine is synchronized to the operation of the state machine of the Timing and Arbitration PAL device by using signal SIG. This PAL device has no function for a refresh cycle. For a processor cycle it asserts the appropriate CAS0-CAS3 and WE signals and generates the cycle complete signals DSACK0–DSACK1. The state machine functions are implemented in the four buried state registers on this device.

CAS Generation

The state machine starts at the initial polling state 0, and remains there until it receives the SIG signal from the timing and arbitration PAL device indicating the start of a processor cycle. The maximum allowable data width for the 68020 processor is 32 bits. The data bus is split into four byte wide segments. The CAS signals for these four byte segments are independently controlled, allowing independent byte-wide read and write capability. These four CAS signals allow read and write for low-low, low-middle, high-middle and high-high bytes of a 32-bit data port. For a 16-bit data port size, CAS 2–3 provide the low and high byte strobes, respectively. For an 8-bit port, only CAS3 is used.

In state 1 the CAS signals are asserted. The CAS signals asserted depend upon the PORT size available, the size of data being transferred. and the alignment of the transfer. As explained before, the attempted transfer data size is encoded on two SIZ0,



441 05

NOTE: WHERE SIGNAL DESIGNATIONS APPEAR BOTH ABOVE AND BELOW A LINE, THE UPPER INDICATES INPUTS AND THE LOWER, OUTPUTS.

* OUTPUTS INDEPENDENT OF RW AND RMC

Figure 5. State Diagram of Data Sizing and Alignment PAL Device

SIZ1 bits by the processor, and the alignment is indicated by the least significant address bits A0, A1 of the processor.

The port size is very application dependent. Different systems might require either static or dynamic selection of this port size. For example, the designer can define different address ranges where the port size may be different. Alternatively, jumpers can also define the port size. A general-purpose means for this has been selected, in the form of two input pins PRT0 & PRT1, which define the size of the port.

Port Size

The size of the data bus is decided based on the input pins PRT1, PRT0.

PRT0	PRT1	SIZE
L	L	8 bits
L	H	16 bits
H	L	16 bits
H	H	32 bits

For 32-bit ports, all four CAS signals need to be asserted. For a 16-bit port only two CAS signals need to be asserted. For an eight-bit port only CAS3 is asserted. As mentioned before, the processor always attempts to transfer the maximum possible amount of data. Its SIZ0-1 signals encode the amount of data being transferred. The table in Figure 6 illustrates the relationship of the data size, port size and alignment which is decided by address bits A1, A0.

The data from this table is directly used in the state description syntax to generate the CAS signals for state 1. The software compiles the Boolean equations for this logic directly.

Once asserted, the CAS signals are held by the state machine until the appropriate read, write, or read-modify-write cycles are complete.

DSACK Acknowledge to CPU

The amount of data receivable by any device is dependent upon its port size. Based on the port size (PRT0-1) the encodings for DSACK0-1 are generated to acknowledge the CPU and inform it of the port size. The timing of acknowledge is synchronized by the state machine. For read and write cycles, the DSACK signals are generated in state 1 and removed on cycle completion in state 4.

For a read-modify-write cycle, a complication exists. The read-modify-write cycle requires two DSACK signals; one for read and one for a write. The first DSACK is generated in state 1 and removed in either state 4 or state 5. The second is generated in state 8 and removed in state 11.

WE Generation

In state 1 the WE is generated for normal write cycles. For read-modify-write cycles, a delayed WE signal is generated based upon the internal four-bit state counter in state 8 (Figure 3).

TRANSFER SIZE	SIZ1	SIZ0	A1	A0	DATA BUS ACTIVE SECTIONS BYTE (B)–WORD (W)–LONG WORD (L) PORTS			
					D31–D24	D23–D16	D15–D8	D7–D0
BYTE	0	1	0	0	BWL	–	–	–
	0	1	0	1	B	WL	–	–
	0	1	1	0	BW	–	L	–
	0	1	1	1	B	W	–	L
WORD	1	0	0	0	BWL	WL	–	–
	1	0	0	1	B	WL	L	–
	1	0	1	0	BW	W	L	L
	1	0	1	1	B	W	–	L
THREE-BYTE	1	1	0	0	BWL	WL	L	–
	1	1	0	1	B	WL	L	L
	1	1	1	0	BW	W	L	L
	1	1	1	1	B	W	–	L
LONG WORD	0	0	0	0	BWL	WL	L	L
	0	0	0	1	B	WL	L	L
	0	0	1	0	BW	W	L	L
	0	0	1	1	B	W	–	L

Figure 6. Data Bus Activity for Byte, Word and Long Word Ports

```
NAME     MEGABIT DRAM TIMING & ARBITRATION CONTROLLER;
PARTNO   AmPAL 23S8 ;
DATE     03/24/86 ;
REV      01 ;
DESIGNER KAPIL SHANKAR ;
COMPANY  ADVANCED MICRO DEVICES ;
ASSEMBLY NONE ;
LOCATION SUNNYVALE, CA ;
DEVICE   P23S8 ;

/*****************************************************************/
/*                                                               */
/*                                                               */
/*                                                               */
/*****************************************************************/
/*  Allowable Target Device Types:  THE ONLY ONE                 */
/*****************************************************************/

/**  Inputs  **/

PIN  2      =  MREQ ;
PIN  3      =  ADSEL0 ;
PIN  4      =  ADSEL1 ;
PIN  5      =  RMC ;
PIN  6      =  RFCLK ;
PIN  7      =  INIT ;

/**  Outputs  **/

PIN  19     = ROWADD ;
PIN  18     = !RAS0 ;
PIN  17     = !COLADD ;
PIN  16     = !RAS1 ;
PIN  15     = !RAS2 ;
PIN  14     = !RFSH ;
PIN  13     = !RAS3 ;
PIN  12     = !SIG ;

NODE  [B0,B1,B2,B3,B4,B5] ;

/** Declarations and Intermediate Variable Definitions **/


FIELD STMA = [B0,B1,B2,B3,B4] ;          /** STATE MACHINE A **/
FIELD STMB = [B5] ;                      /** STATE MACHINE B **/
FIELD BANK = [ADSEL1, ADSEL0] ;          /** MEMORY BANK ADDRESSED **/

BANK0 = BANK : 0 ;
BANK1 = BANK : 1 ;
BANK2 = BANK : 2 ;
BANK3 = BANK : 3 ;

$DEFINE  SA0  'b'00000                   /** STATE 0 **/
$DEFINE  SA1  'b'00001                   /** STATE 1 **/
$DEFINE  SA2  'b'00010                   /** STATE 2 **/
$DEFINE  SA3  'b'00011                   /** STATE 3 **/
$DEFINE  SA4  'b'00100                   /** STATE 4 **/
$DEFINE  SA5  'b'00101                   /** STATE 5 **/
$DEFINE  SA6  'b'00110                   /** STATE 6 **/
$DEFINE  SA7  'b'00111                   /** STATE 7 **/
$DEFINE  SA8  'b'01000                   /** STATE 8 **/
$DEFINE  SA9  'b'01001                   /** STATE 9 **/
$DEFINE  SA10  'b'01010                  /** STATE 10 **/
$DEFINE  SA11  'b'01011                  /** STATE 11 **/
$DEFINE  SA12  'b'01100                  /** STATE 12 **/
$DEFINE  SA13  'b'01101                  /** STATE 13 **/
$DEFINE  SA14  'b'01110                  /** STATE 14 **/
$DEFINE  SA15  'b'01111                  /** STATE 15 **/
$DEFINE  SA16  'b'10000                  /** STATE 16 **/
$DEFINE  SA17  'b'10001                  /** STATE 17 **/
$DEFINE  SA18  'b'10010                  /** STATE 18 **/
$DEFINE  SA19  'b'10011                  /** STATE 19 **/
$DEFINE  SA20  'b'10100                  /** STATE 20 **/
$DEFINE  SA21  'b'10101                  /** STATE 21 **/
$DEFINE  SA22  'b'10110                  /** STATE 22 **/
```

Figure 7. Source Code for Timing and Controller PAL Device

```
$DEFINE  SB0  'b'0                    /** STATE 0 **/
$DEFINE  SB1  'b'1                    /** STATE 1 **/

/**  Logic Equations  **/

/** THE FIRST STATE MACHINE **/

SEQUENCE STMA {

/** POLLING FOR PROCESSOR OR REFRESH CYCLES **/

PRESENT SA0  IF !B5 & !MREQ & BANK0       NEXT SA1  OUT RAS0 ;
             IF !B5 & !MREQ & BANK1       NEXT SA1  OUT RAS1 ;
             IF !B5 & !MREQ & BANK2       NEXT SA1  OUT RAS2 ;
             IF !B5 & !MREQ & BANK3       NEXT SA1  OUT RAS3 ;
/** PROCESSOR CYCLE **/
/** JUMP TO STATE SA1 **/
/** ASSERT THE RAS FOR THE MEMORY BANK ADDRESSED **/

             IF  B5                       NEXT SA16 OUT ROWADD
                                               OUT RFSH ;
/** REFRESH CYCLE **/
/** JUMP TO STATE SA16 **/
/** REMOVE ROW ADDRESS AND SET UP REFRESH ADDRESS **/

             DEFAULT                      NEXT SA0  ;
/** ELSE POLL AGAIN **/
/** STAY IN STATE SA0 **/


/** THIS IS THE PROCESSOR CYCLE **/

PRESENT SA1  IF BANK0                     NEXT SA2  OUT RAS0 ;
             IF BANK1                     NEXT SA2  OUT RAS1 ;
             IF BANK2                     NEXT SA2  OUT RAS2 ;
             IF BANK3                     NEXT SA2  OUT RAS3 ;
                                          NEXT SA2  OUT ROWADD
                                                    OUT COLADD
                                                    OUT SIG ;
/** REMOVE ROW ADDRESS AND SET UP COLUMN ADDRESS **/
/** ASSERT SIG TO INFORM DATA SIZING & ALIGNMENT PAL OF PROCESSOR CYCLE **/

PRESENT SA2  IF BANK0                     NEXT SA3  OUT RAS0 ;
             IF BANK1                     NEXT SA3  OUT RAS1 ;
             IF BANK2                     NEXT SA3  OUT RAS2 ;
             IF BANK3                     NEXT SA3  OUT RAS3 ;
                                          NEXT SA3  OUT ROWADD
                                                    OUT COLADD
                                                    OUT SIG ;
/** MEMORY ACCESS TIME DELAY **/

PRESENT SA3  IF BANK0                     NEXT SA4  OUT RAS0 ;
             IF BANK1                     NEXT SA4  OUT RAS1 ;
             IF BANK2                     NEXT SA4  OUT RAS2 ;
             IF BANK3                     NEXT SA4  OUT RAS3 ;
                                          NEXT SA4  OUT ROWADD
                                                    OUT COLADD
                                                    OUT SIG ;

PRESENT SA4  IF BANK0 & RMC               NEXT SA5  OUT !RAS0 ;
             IF BANK1 & RMC               NEXT SA5  OUT !RAS1 ;
             IF BANK2 & RMC               NEXT SA5  OUT !RAS2 ;
             IF BANK3 & RMC               NEXT SA5  OUT !RAS3 ;
             IF RMC                       NEXT SA5  OUT !ROWADD
                                                    OUT !COLADD
                                                    OUT SIG ;
/** DISTINGUISH BETWEEN READ WRITE OR READ MODIFY WRITE CYCLES **/
/** IF RMC HIGH EXECUTE AS READ WRITE CYCLE **/
/** JUMP TO STATE SA5 **/
/** ALSO REMOVE COLUMN ADDRESS AND APPLY ROW ADDRESS **/
/** ALSO REMOVE THE RAS SIGNAL **/

             IF BANK0 & !RMC              NEXT SA8  OUT RAS0 ;
             IF BANK1 & !RMC              NEXT SA8  OUT RAS1 ;
             IF BANK2 & !RMC              NEXT SA8  OUT RAS2 ;
             IF BANK3 & !RMC              NEXT SA8  OUT RAS3 ;
             IF !RMC                      NEXT SA8  OUT !ROWADD
                                                    OUT !COLADD
                                                    OUT SIG ;
/** IF RMC LOW EXECUTE READ MODIFY WRITE CYCLE **/
/** JUMP TO STATE SA8 **/
/** ALSO REMOVE COLUMN ADDRESS AND ASSERT ROW ADDRESS **/
```

Figure 7. Source Code for Timing and Controller PAL Device (Cont'd.)

```
PRESENT SA5                              NEXT SA6  OUT !SIG ;
/** RAS PRECHARGE TIME DELAY **/

PRESENT SA6                              NEXT SA7  ;

PRESENT SA7                              NEXT SA0  ;
/** PROCESSOR READ WRITE CYCLE COMPLETE **/


/** THIS IS THE READ MODIFY WRITE CYCLE **/

PRESENT SA8  IF BANK0                    NEXT SA9  OUT RAS0 ;
             IF BANK1                    NEXT SA9  OUT RAS1 ;
             IF BANK2                    NEXT SA9  OUT RAS2 ;
             IF BANK3                    NEXT SA9  OUT RAS3 ;
                                         NEXT SA9  OUT SIG  ;
/** ACCESS TIME DELAY FOR BOTH READ AND WRITE **/

PRESENT SA9  IF BANK0                    NEXT SA10 OUT RAS0 ;
             IF BANK1                    NEXT SA10 OUT RAS1 ;
             IF BANK2                    NEXT SA10 OUT RAS2 ;
             IF BANK3                    NEXT SA10 OUT RAS3 ;
                                         NEXT SA10 OUT SIG  ;

PRESENT SA10 IF BANK0                    NEXT SA11 OUT RAS0 ;
             IF BANK1                    NEXT SA11 OUT RAS1 ;
             IF BANK2                    NEXT SA11 OUT RAS2 ;
             IF BANK3                    NEXT SA11 OUT RAS3 ;
                                         NEXT SA11 OUT SIG  ;

PRESENT SA11 IF BANK0                    NEXT SA12 OUT RAS0 ;
             IF BANK1                    NEXT SA12 OUT RAS1 ;
             IF BANK2                    NEXT SA12 OUT RAS2 ;
             IF BANK3                    NEXT SA12 OUT RAS3 ;
                                         NEXT SA12 OUT SIG  ;

PRESENT SA12 IF BANK0                    NEXT SA13 OUT RAS0 ;
             IF BANK1                    NEXT SA13 OUT RAS1 ;
             IF BANK2                    NEXT SA13 OUT RAS2 ;
             IF BANK3                    NEXT SA13 OUT RAS3 ;
                                         NEXT SA13 OUT SIG  ;
```

```
/** THIS IS THE READ WRITE CYCLE **/

PRESENT SA13 IF BANK0                    NEXT SA14 OUT !RAS0 ;
             IF BANK1                    NEXT SA14 OUT !RAS1 ;
             IF BANK2                    NEXT SA14 OUT !RAS2 ;
             IF BANK3                    NEXT SA14 OUT !RAS3 ;
                                         NEXT SA14 OUT SIG  ;
/** REMOVE THE RAS SIGNAL **/

PRESENT SA14                             NEXT SA15 OUT !SIG ;
/** RAS PRECHARGE TIME DELAY **/

PRESENT SA15                             NEXT SA0 ;

/** PROCESSOR READ MODIFY WRITE CYCLE COMPLETE **/


/** THIS IS THE MEMORY REFRESH CYCLE **/

PRESENT SA16                             NEXT SA17 OUT RAS0
                                                   OUT RAS1
                                                   OUT RAS2
                                                   OUT RAS3
                                                   OUT ROWADD
                                                   OUT RFSH ;
/** ASSERT ALL FOUR RAS SIGNALS **/

PRESENT SA17                             NEXT SA18 OUT RAS0
                                                   OUT RAS1
                                                   OUT RAS2
                                                   OUT RAS3
                                                   OUT ROWADD
                                                   OUT RFSH ;
/** REFRESH TIME DELAY **/

PRESENT SA18                             NEXT SA19 OUT RAS0
                                                   OUT RAS1
                                                   OUT RAS2
                                                   OUT RAS3
                                                   OUT ROWADD
                                                   OUT RFSH ;
```

Figure 7. Source Code for Timing and Controller PAL Device (Cont'd.)

```
PRESENT SA19                          NEXT SA20 OUT !RAS0
                                                OUT !RAS1
                                                OUT !RAS2
                                                OUT !RAS3
                                                OUT !ROWADD
                                                OUT !RFSH ;

/** REMOVE ALL RAS SIGNALS **/
/** ALSO REMOVE REFRESH ADDRESS AND ASSERT ROW ADDRESS **/

PRESENT SA20                          NEXT SA21 ;
/** RAS PRECHARGE TIME DELAY **/

PRESENT SA21                          NEXT SA22 ;

PRESENT SA22                          NEXT SA0  ;

/** TERMINATE MEMORY REFRESH CYCLE **/
/** GO BACK TO POLLING STATE **/
}
/** THE FIRST STATE MACHINE COMPLETE **/

/** THE SECOND STATE MACHINE **/

SEQUENCE STMB {

PRESENT SB0  IF RFCLK                 NEXT SB1 ;
             DEFAULT                  NEXT SB0 ;
/** JUMP TO STATE SB1 **/
/** REMEMBER REFRESH REQUEST **/

PRESENT SB1  IF RFSH                  NEXT SB0 ;
             DEFAULT                  NEXT SB1 ;
/** JUMP TO STATE SB0 **/
/** REFRESH CYCLE EXECUTED THUS RETURN **/

}

/** THE SECOND STATE MACHINE COMPLETE **/
```

```
/** EQUATIONS FOR SYSTEM INITIALIZATION AFTER RESET **/

[B0..B5].ar = !INIT ;

RAS0.AR    = !INIT ;
RAS1.AR    = !INIT ;
ROWADD.AR  = !INIT ;
COLADD.AR  = !INIT ;
RAS2.AR    = !INIT ;
RAS3.AR    = !INIT ;
RFSH.AR    = !INIT ;
SIG.AR     = !INIT ;

/* THESE ARE THE OUTPUT ENABLES */

RAS0.OE    = 'B'1  ;
RAS1.OE    = 'B'1  ;
ROWADD.OE  = 'B'1  ;
COLADD.OE  = 'B'1  ;
RAS2.OE    = 'B'1  ;
RAS3.OE    = 'B'1  ;
RFSH.OE    = 'B'1  ;
SIG.OE     = 'B'1  ;
```

Figure 7. Source Code for Timing and Controller PAL Device (Cont'd.)

```
NAME     MEGABYT DRAM DATA SIZING & ALIGNMENT CONTROLLER;
PARTNO   AmPAL 23S8 ;
DATE     03/24/86 ;
REV      01 ;
DESIGNER KAPIL SHANKAR ;
COMPANY  ADVANCED MICRO DEVICES ;
ASSEMBLY NONE ;
LOCATION SUNNYVALE, CA ;
DEVICE   P23S8 ;

/*****************************************************************/
/*                                                               */
/*                                                               */
/*                                                               */
/*****************************************************************/
/*  Allowable Target Device Types:  THE ONLY ONE                 */
/*****************************************************************/

/**  Inputs  **/

PIN  2      =  SIG ;
PIN  3      =  A0 ;
PIN  4      =  A1 ;
PIN  5      =  RW ;
PIN  6      =  RMC ;
PIN  7      =  SIZ0 ;
PIN  8      =  SIZ1 ;
PIN  9      =  PRT0 ;
PIN  11     =  PRT1 ;
PIN  12     =  INIT ;

/**  Outputs  **/

PIN  19     = !WE ;
PIN  18     = !CAS2 ;
PIN  17     = !CAS0 ;
PIN  16     = !CAS3 ;
PIN  15     = !CAS1 ;
PIN  14     = !DSACK1 ;
PIN  13     = !DSACK0 ;

NODE  [B0,B1,B2,B3] ;

/** Declarations and Intermediate Variable Definitions **/

FIELD STM = [B0,B1,B2,B3] ;            /** THE STATE MACHINE **/

FIELD PORTSIZE = [PRT1, PRT0] ;        /** SIZE OF THE PORT **/
                                       /** INTERFACED TO PROCESSOR **/
PORT8  = PORTSIZE : 0 ;                /**  8 BIT WIDE PORT **/
PORT16 = PORTSIZE : 2 ;                /** 16 BIT WIDE PORT **/
PORT32 = PORTSIZE : 3 ;                /** 32 BIT WIDE PORT **/

/** DEFINE THE STATES  **/

$DEFINE  SA0   'b'0000                 /** STATE 0 **/
$DEFINE  SA1   'b'0001                 /** STATE 1 **/
$DEFINE  SA2   'b'0010                 /** STATE 2 **/
$DEFINE  SA3   'b'0011                 /** STATE 3 **/
$DEFINE  SA4   'b'0100                 /** STATE 4 **/
$DEFINE  SA5   'b'0101                 /** STATE 5 **/
$DEFINE  SA6   'b'0110                 /** STATE 6 **/
$DEFINE  SA7   'b'0111                 /** STATE 7 **/
$DEFINE  SA8   'b'1000                 /** STATE 8 **/
$DEFINE  SA9   'b'1001                 /** STATE 9 **/
$DEFINE  SA10  'b'1010                 /** STATE 10 **/
$DEFINE  SA11  'b'1011                 /** STATE 11 **/

/**  Logic Equations  **/

/** THE STATE MACHINE **/

SEQUENCE STM {

/** POLLING FOR SIG INDICATING A PROCESSOR CYCLE **/
```

Figure 8. Source Code for Data Sizing and Alignment PAL Device

```
/**          PORT    DATA     OFFSET  STATE    OUTPUT  **/
PRESENT SA0
/** FOR PORTSIZE 8 **/
   IF !SIG & PORT8                    NEXT SA1  OUT CAS3 ;

/** GENERATE ACKNOWLEDGE **/
   IF !SIG & PORT8                    NEXT SA1  OUT DSACK0 ;


/** FOR PORTSIZE 16 **/
   IF !SIG & PORT16 & !A0              NEXT SA1  OUT CAS3 ;
   IF !SIG & PORT16 & !SIZ0 & SIZ1 & A0 NEXT SA1  OUT CAS2 ;

/** GENERATE ACKNOWLEDGE **/
   IF !SIG & PORT16                   NEXT SA1  OUT DSACK1 ;


/** FOR PORTSIZE 32 **/
/** BYTE TRANSFER SIZE **/
   IF !SIG & PORT32 & !A1 & !A0           NEXT SA1  OUT CAS3 ;
   IF !SIG & PORT32 & !A1 & A0            NEXT SA1  OUT CAS2 ;
   IF !SIG & PORT32 & !A1 & !SIZ0         NEXT SA1  OUT CAS2 ;
   IF !SIG & PORT32 & !A1 & SIZ1          NEXT SA1  OUT CAS2 ;
   IF !SIG & PORT32 & A1  & !A0           NEXT SA1  OUT CAS1 ;
   IF !SIG & PORT32 & !A1 & !SIZ1 & !SIZ0 NEXT SA1  OUT CAS1 ;
   IF !SIG & PORT32 & !A1 & SIZ1 & SIZ0   NEXT SA1  OUT CAS1 ;
   IF !SIG & PORT32 & !A1 & A0 & !SIZ0    NEXT SA1  OUT CAS1 ;
   IF !SIG & PORT32 & A1 & A0             NEXT SA1  OUT CAS3 ;
   IF !SIG & PORT32 & !SIZ1 & !SIZ0       NEXT SA1  OUT CAS2 ;
   IF !SIG & PORT32 & A0 & SIZ1 & SIZ0    NEXT SA1  OUT CAS1 ;

/** GENERATE ACKNOWLEDGE **/
   IF !SIG & PORT32                   NEXT SA1  OUT DSACK0
                                                OUT DSACK1 ;

/** JUMP TO STATE SA1 **/
/** ASSERT THE CAS FOR THE RIGHT PORTSIZE DATASIZE AND ADDRESS OFFSET **/

   IF !SIG & !RW & RMC                NEXT SA1  OUT WE ;
/** GENERATE WE FOR READ WRITE CYCLE ONLY **/

   IF SIG                             NEXT SA0  ;
```

```
/** ELSE WAIT FOR THE PROCESSOR CYCLE **/
/** POLL AGAIN **/
/** STAY IN STATE SA0 **/

/** THIS IS THE PROCESSOR CYCLE START **/

PRESENT SA1
   IF CAS0                            NEXT SA2  OUT CAS0 ;
   IF CAS1                            NEXT SA2  OUT CAS1 ;
   IF CAS2                            NEXT SA2  OUT CAS2 ;
   IF CAS3                            NEXT SA2  OUT CAS3 ;
   IF !RW & RMC                       NEXT SA2  OUT WE ;
   IF PORT8                           NEXT SA2  OUT DSACK0 ;
   IF PORT16                          NEXT SA2  OUT DSACK1 ;
   IF PORT32                          NEXT SA2  OUT DSACK0
                                                OUT DSACK1 ;
                                      NEXT SA2 ;
/** HOLD CAS GENERATED IN THE LAST STATE **/
/** IF WRITE CYCLE AND NOT READ MODIFY WRITE CYCLE ASSERT WE **/
/** JUMP TO STATE SA2 **/

PRESENT SA2
   IF CAS0                            NEXT SA3  OUT CAS0 ;
   IF CAS1                            NEXT SA3  OUT CAS1 ;
   IF CAS2                            NEXT SA3  OUT CAS2 ;
   IF CAS3                            NEXT SA3  OUT CAS3 ;
   IF !RW & RMC                       NEXT SA3  OUT WE ;
   IF PORT8                           NEXT SA3  OUT DSACK0 ;
   IF PORT16                          NEXT SA3  OUT DSACK1 ;
   IF PORT32                          NEXT SA3  OUT DSACK0
                                                OUT DSACK1 ;
                                      NEXT SA3 ;

/** HOLD CAS GENERATED IN THE LAST STATE **/
/** HOLD WE FOR ACCESS DURATION **/
/** JUMP TO STATE SA3 **/

PRESENT SA3
   IF RMC                             NEXT SA4  OUT !CAS0
                                                OUT !CAS1
                                                OUT !CAS2
```

Figure 8. Source Code for Data Sizing and Alignment PAL Device (Cont'd.)

```
                                        OUT !CAS3
                                        OUT !WE
                                        OUT !DSACK0
                                        OUT !DSACK1 ;

/** FOR READ WRITE CYCLE REMOVE CAS AND WE FOR CYCLE COMPLETION **/
/** ACKNOWLEDGE CYCLE COMPLETION BY ACKNOT **/
/** JUMP TO STATE SA4 **/

   IF CAS0 & !RMC                NEXT SA5  OUT CAS0 ;
   IF CAS1 & !RMC                NEXT SA5  OUT CAS1 ;
   IF CAS2 & !RMC                NEXT SA5  OUT CAS2 ;
   IF CAS3 & !RMC                NEXT SA5  OUT CAS3 ;
   IF !RMC                       NEXT SA5  OUT !DSACK0
                                           OUT !DSACK1 ;

/** FOR READ MODIFY WRITE CYCLE CONTINUE TO ASSERT CAS SIGNALS **/
/** ACKNOWLEDGE COMPLETION OF READ PORTION OF THE CYCLE BY ACKNOT **/
/** JUMP TO STATE SA5 **/


/** REMAINING PORTION OF THE READ WRITE CYCLE **/

PRESENT SA4                          NEXT SA0 ;
/** GO BACK TO POLLING STATE SA0 **/

/** COMPLETION OF READ WRITE CYCLE **/




/** REMAINING PORTION OF THE READ MODIFY WRITE CYCLE **/

PRESENT SA5
   IF CAS0                       NEXT SA6  OUT CAS0 ;
   IF CAS1                       NEXT SA6  OUT CAS1 ;
   IF CAS2                       NEXT SA6  OUT CAS2 ;
   IF CAS3                       NEXT SA6  OUT CAS3 ;
                                 NEXT SA6 ;

/** CONTINUE THE READ MODIFY WRITE CYCLE **/

PRESENT SA6
   IF CAS0                                 NEXT SA7  OUT CAS0 ;
   IF CAS1                                 NEXT SA7  OUT CAS1 ;
   IF CAS2                                 NEXT SA7  OUT CAS2 ;
   IF CAS3                                 NEXT SA7  OUT CAS3 ;
                                           NEXT SA7 ;

/** CONTINUE THE READ MODIFY WRITE CYCLE **/

PRESENT SA7
   IF CAS0                                 NEXT SA8  OUT CAS0 ;
   IF CAS1                                 NEXT SA8  OUT CAS1 ;
   IF CAS2                                 NEXT SA8  OUT CAS2 ;
   IF CAS3                                 NEXT SA8  OUT CAS3 ;
   IF PORT8                                NEXT SA8  OUT DSACK0 ;
   IF PORT16                               NEXT SA8  OUT DSACK1 ;
   IF PORT32                               NEXT SA8  OUT DSACK0
                                                     OUT DSACK1 ;
                                           NEXT SA8  OUT WE ;

/** THIS IS THE WRITE PORTION OF THE READ MODIFY WRITE CYCLE **/
/** GENERATE THE WE SIGNAL FOR ENABLING WRITE **/
/** GENERATE THE CORRECT ACKNOWLEDGE FOR THE WRITE CYCLE PORTION **/
/** CONTINUE TO ASSERT CAS **/


PRESENT SA8
   IF CAS0                                 NEXT SA9  OUT CAS0 ;
   IF CAS1                                 NEXT SA9  OUT CAS1 ;
   IF CAS2                                 NEXT SA9  OUT CAS2 ;
   IF CAS3                                 NEXT SA9  OUT CAS3 ;
   IF PORT8                                NEXT SA9  OUT DSACK0 ;
   IF PORT16                               NEXT SA9  OUT DSACK1 ;
   IF PORT32                               NEXT SA9  OUT DSACK0
                                                     OUT DSACK1 ;
                                           NEXT SA9  OUT WE ;

/** THIS IS THE WRITE PORTION OF THE READ MODIFY WRITE CYCLE **/
```

Figure 8. Source Code for Data Sizing and Alignment PAL Device (Cont'd.)

```
PRESENT SA9
   IF CAS0                    NEXT SA10 OUT CAS0 ;
   IF CAS1                    NEXT SA10 OUT CAS1 ;
   IF CAS2                    NEXT SA10 OUT CAS2 ;
   IF CAS3                    NEXT SA10 OUT CAS3 ;
   IF PORT8                   NEXT SA10 OUT DSACK0 ;
   IF PORT16                  NEXT SA10 OUT DSACK1 ;
   IF PORT32                  NEXT SA10 OUT DSACK0
                                        OUT DSACK1 ;
                              NEXT SA10 OUT WE ;

/** THIS IS THE WRITE PORTION OF THE READ MODIFY WRITE CYCLE **/

PRESENT SA10
                              NEXT SA11 OUT !CAS0
                                        OUT !CAS1
                                        OUT !CAS2
                                        OUT !CAS3
                                        OUT !DSACK0
                                        OUT !DSACK1
                                        OUT !WE ;

/** COMPLETION OF THE WRITE PORTION OF THE READ MODIFY WRITE CYCLE **/
/** REMOVE ALL CAS  WE  AND ACKNOWLEDGE COMPLETION **/

/** REMAINING PORTION OF THE READ MODIFY WRITE CYCLE **/

PRESENT SA11                    NEXT SA0 ;
/** GO BACK TO POLLING STATE SA0 **/
/** COMPLETION OF THE READ MODIFY WRITE CYCLE **/
}
/** THE STATE MACHINE COMPLETE **/
```

```
/** EQUATIONS FOR SYSTEM INITIALIZATION AFTER RESET **/

[B0..B3].ar = !INIT ;

CAS0.AR    = !INIT ;
CAS1.AR    = !INIT ;
DSACK0.AR  = !INIT ;
DSACK1.AR  = !INIT ;
CAS2.AR    = !INIT ;
CAS3.AR    = !INIT ;
WE.AR      = !INIT ;

/* THESE ARE THE OUTPUT ENABLES */

CAS0.OE    = 'B'1  ;
CAS1.OE    = 'B'1  ;
DSACK0.OE  = 'B'1  ;
DSACK1.OE  = 'B'1  ;
CAS2.OE    = 'B'1  ;
CAS3.OE    = 'B'1  ;
WE.OE      = 'B'1  ;
```

Figure 8. Source Code for Data Sizing and Alignment PAL Device (Cont'd.)

8088 To Am2968 Interface

Introduction

This application note describes the implementation of a timing generator which interfaces between the 8088 and the Am2968 Dynamic Memory Controller (DMC) using programmable logic and a delay line. The implementation does not account for any error detection and correction (EDC) circuitry.

The timing generator, like the Am2970, is needed in memory systems utilizing a dynamic memory controller. It serves as an interface between the processor and the controller and generates the necessary control signals for the controller.

A dynamic memory controller, in brief, interfaces between a processor and the dynamic memory array. It steers the appropriate address inputs and Row and Column Address Strobes ($\overline{RAS}$ and $\overline{CAS}$) required in the selected memory operations (i.e. refreshing, read/write).

The following sections will introduce and illustrate the implementation of the timing generator using 20-pin PAL devices, and also show its interface to AMD's Am2968 (DMC) and 8088 processor. Figure 1 shows the block diagram of the interface. Programmable logic was chosen because it is readily available, simple to use and reduces the number of devices required to implement specific functions.

Interface Overview

The implementation of this timing generator is not a general purpose application. It is dedicated specifically to a particular processor, the Intel 8088, and is limited to accessing only four banks of memory. The range of the four banks of memory is therefore 1 Mbyte—with 256K in each bank. This limitation is set by the fact that no additional decoding circuitry has been added. Decoding is done with the two address lines (A18–A19) of the 8088, thus allowing only four banks to be selected.

The clock to the 8088 and the two PAL devices is provided by the 8284A. The CLK signal from the 8284A is connected directly to the CLK input of the 8088 and also to CLK of RASER (PAL16R4). The PCLK of the 8284A connects to the PCLK input of HLDR (PAL16R6) to operate the 6-bit counter. Note that the clock operates on a 1/3 and 2/3 duty cycle.

Description of PAL Device Function

The function of RASER (PAL16R4) is to create RASI (Row Address Strobe) and CASI (Column Address Strobe) at the appropriate time. RASI is generated directly from the device, whereas CASI is generated from MRASI via a delay line. Figure 2 shows the timing delays between the controller and processor interface signals. The two modes of operation which the timing is focused on is Refresh w/o Scrubbing and Read/Write. By toggling the state of the mode control pin MC1 to either a LOW or HIGH respectively, and with MC0 tied LOW (at the Am2968), the desired mode is generated. The mode control pin MC1 is generated based on the state of the processor's interface pins: IO/$\overline{M}$, DT/$\overline{R}$ and $\overline{SS0}$. The combination of these signals decodes the processor's current bus cycle and indicates the ongoing memory activity. The MUX Select pin (MSEL) is also generated from the delay line. It determines whether a Row or Column address is sent to the memory address input based on the mode control inputs, MC0 and MC1.

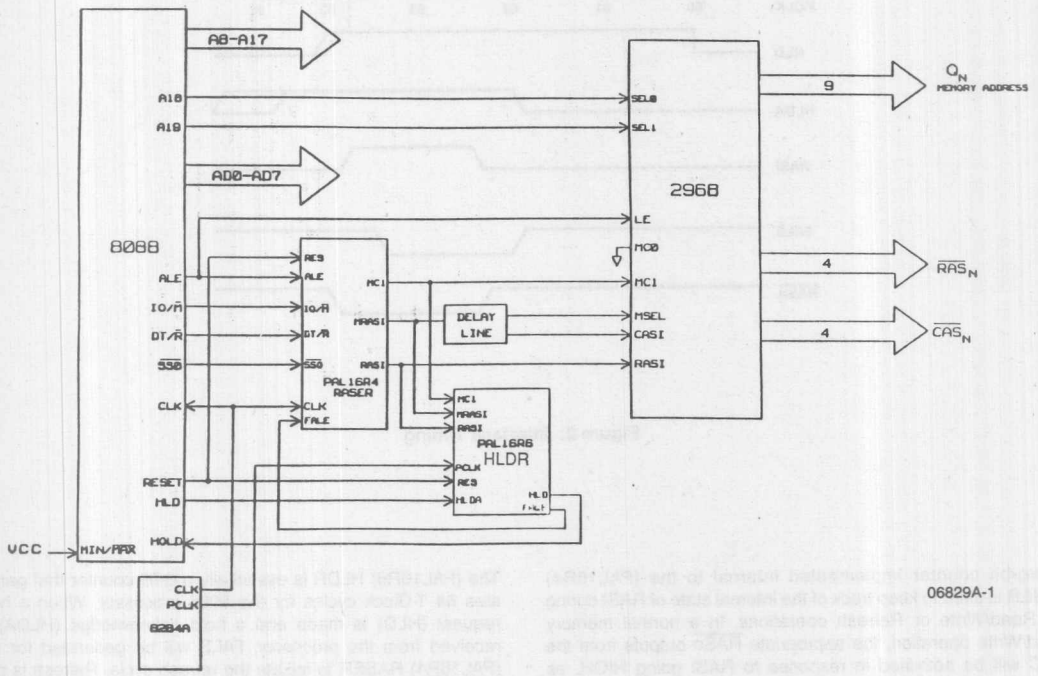


Figure 1. Am2968 to 8088 Interface Block Diagram

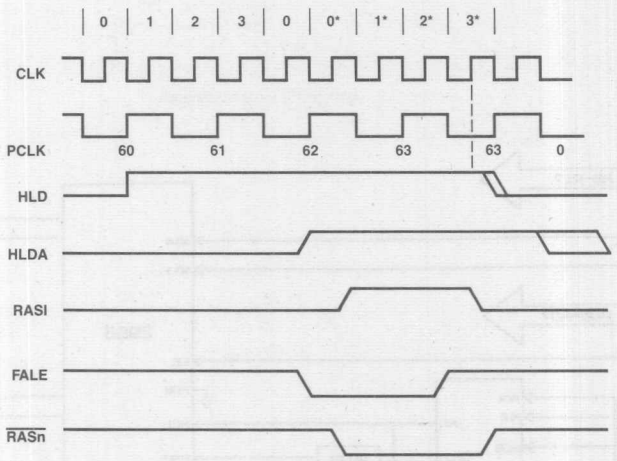


Figure 2. Interface Timing

A two-bit counter implemented internal to the (PAL16R4) RASER is used to keep track of the internal state of RASI during the Read/Write or Refresh operations. In a normal memory Read/Write operation, the appropriate $\overline{\text{RASn}}$ outputs from the DMC will be activated in response to RASI going HIGH, as shown in the memory timing in Figure 3, during this time ALE initiates the cycle. In Refresh mode, receiving RASI will force all the $\overline{\text{RASn}}$ outputs of the DMC to go Low. FALE initiates the refresh cycle, since no ALE occurs during this time, as shown in refresh timing of Figure 3. The counter is also configured such that, during memory operations, sufficient time has been allotted within the cycle for $\overline{\text{RAS}}$ precharge (required in DRAMs) to occur.

b) HLDR (PAL16R6)

The function of the (PAL16R6) HLDR is to generate the two signals—FALE (False Address Latch Enable) and HLD (Hold). FALE controls the internal latches (ALS and ALR) of the (PAL16R4) RASER and initiates the refresh cycle independent of the processor's ALE (Address Latch Enable) signal.

The (PAL16R6) HLDR is essentially a 6-bit counter that generates 64 T-Clock cycles for the 8088 processor. When a hold request (HLD) is made and a hold acknowledge (HLDA) is received from the processor, FALE will be generated for the (PAL16R4) RASER to initiate the refresh cycle. Refresh is performed every 8 μs which is double the required frequency. This time is derived in the Refresh Calculations section of this application note. The 8088 processor allows request only at the end of the bus cycle. A bus cycle duration is 4 T-Clocks, thus refresh is performed at cycles "60–63" as noted in the function table in the HLDR design specification. Figure 2 refresh timing shows this critical portion of the timing activity when RASI is generated for refresh. During CLK (0 - 0*), (this corresponds to PCLK count value 62), the processor acknowledge is received and RASI becomes active. While RASI is HIGH, the internal 2-bit counter will cycle through its count sequence (0* - 3*); at the end of the sequence as indicated by PCLK count value 63 (i.e., CLK 3*), RASI will become inactive.

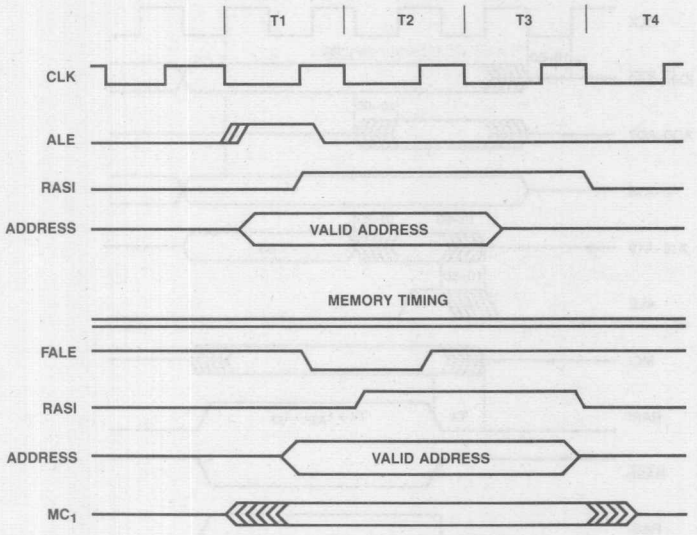


Figure 3. Refresh Timing

Refresh Timing Calculations

The timing calculations are based on 8088 processor being in minimum mode (8 MHz) and a cell array of 128 rows.

The 8088 clock period is 8 MHz, thus

t = 1/f = 1/8 M = 125 ns.

Refreshing for DRAMs are performed on 128 rows every 2 ms. The calculations for the above memory array is determined to be:

Required refresh time = 2 ms/128 = 15.6 μs.

Thus the required time for 256 row cell array will be:

Required refresh time = 4 ms/256 = 15.6 μs.

From the above calculations, refreshing needs to be performed at least every 15 μs. But for this application, memory is being refreshed every 8 μs as shown below:

Refresh time = 64 T-clock cycles x 125 ns = 8 μs.

Delay Line Taps Computation

The calculations for the assignment of the delay line taps is based on the parameters CASI and MSEL relative to RASI making a transition to the active state. See the timing in Figure 4.

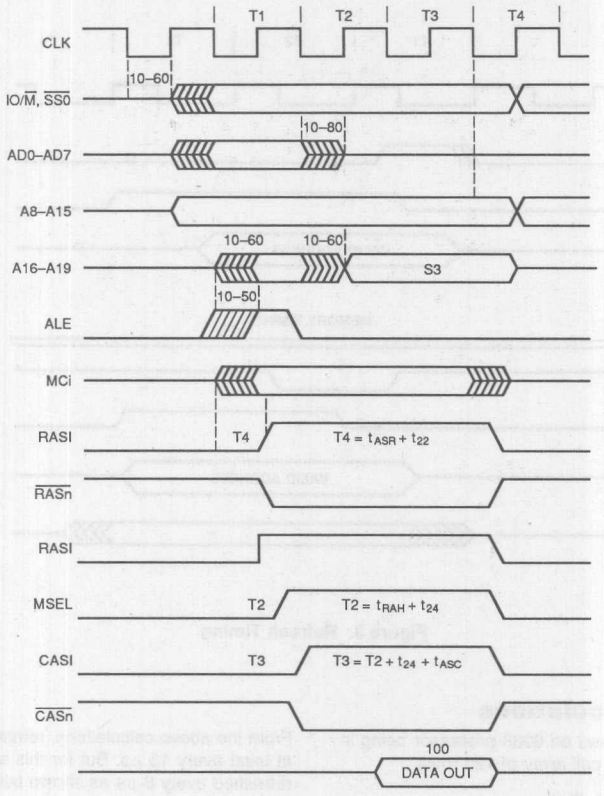


Figure 4. Delay Line Timing Diagram

MSEL may switch some time after T_2. T_2 is calculated to be as follows:

$T_2 = t_{RAH} + t_{24} = 20 + 11 = 31$ ns.

where T_2 = delay time from RASI to MSEL
t_{RAH} = Row Address Hold Time of the DRAM
t_{24} = skew time for Qn to $\overline{RASn}$ of the DMC

Thus, after 31 ns MSEL should initiate its active transition.

CASI is calculated to make its change; T_3 ns relative to RASI. The calculated value is as follows:

$T_3 = T_2 + t_{25} + t_{ASC} = 31 + 33 + 0 = 64$.

where T_3 = delay from RASI to $\overline{CASI}$
t_{25} = skew time for Qn to $\overline{CASn}$ of the DMC
t_{ASC} = column setup time of the DRAM

Thus, 64 ns after RASI initiates its transition, CASI may begin to go active.

For simplicity, delay line taps may be assigned as follows:

RASI ---------- MSEL ~ 30 ns
RASI ---------- CASI ~ 60 ns

Counter and Mode Function Table

The following function tables describe the activity of the RASI/MRASI during the specific modes of the processor's interface signals (IO/$\overline{M}$, DT/$\overline{R}$ and $\overline{SS0}$) and the state of the internal counter which controls RASI/MRASI. Note: RASI will be active during the counter sequences between 1 to 3 as shown below in Figure 5.

2

Two Bit Counter

A	B		RASI
0	0		0
•	•		•
•	•		•
0	0	;0	0
1	0	;1	1
0	1	;2	1
1	1	;3	1

Two Bit Counter

Sequence		IO/$\overline{M}$	DT/$\overline{R}$	$\overline{SS0}$	MRASI	COMMENTS
0		1	0	0 ------	0 ------	IACK
0		1	0	1 ------	0 ------	IOR
0		1	1	0 ------	0 ------	IOW
0		1	1	1 ------	0 ------	HALT
0		0	1	1 ------	0 ------	PASSIVE
1	3 ------	0	0	0 ------	1 ------	CODE ACCESS
1	3 ------	0	0	1 ------	1 ------	RD
1	3 ------	0	1	0 ------	1 ------	WR

Note: This design does not allow the user to insert Wait States during memory operations! Wait States may be inserted in interrupt or I/O cycles.

Figure 5.

```
TITLE     Am2968 TO 8088 INTERFACE
PATTERN   RASER
REVISION  01
AUTHOR    G. SPEARS
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP RASER PAL16R4

CLK   RES   ALE   FALE   IOM   DTR   SS0   CLK2   NU2   GND
TRI   MRASI   RASI   /MC1   NC   /B   /A   /ALR   /ALS   VCC

EQUATIONS

ALS =   FALE*/ALE*/ALR
      + RES

ALR = /A*B*/ALS
      + A*/ALS

/RASI = /A*/B*/RES
        + RES*CLK2

/MRASI = IOM*/DTR
         + IOM*DTR
         + /IOM*DTR*SS0
         + /A*/B

A := /A*/RES*/ALS
   + /A*/RES*ALE

B := A*/B*/RES*/ALS
   + /A*B*/RES*/ALS

MC1 := IOM*/DTR*ALE*/RES
     + DTR*SS0*ALE*/RES
     + IOM*DTR*/SS0*ALE*/RES
     + MC1*/ALE*/RES

; SIMULATION NOT INCLUDED
```

Figure 6. Source Listing for the 8088 to Am2968 Interface (AmPAL16R4)

```
TITLE     Am2968 TO 8088 INTERFACE
PATTERN   HLDR
REVISION  01
AUTHOR    G. SPEARS
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

. CHIP HLDR PAL16R6

PCLK RASI MRASI HLDA RES CLK MC1 NU2 NU3 GND
TRI /FALE /F /E /D /C /B /A HLD VCC

EQUATIONS

A := /A*/RES*MC1

B := A*/B*/RES*MC1
   + /A*B*/RES*MC1

C := A*B*/C*/RES*MC1
   + /B*C*/RES*MC1
   + /A*C*/RES*MC1

D := A*B*C*/D*/RES*MC1
   + /C*D*/RES*MC1
   + /B*D*/RES*MC1
   + /A*D*/RES*MC1

E :=  A*B*C*D*/E*/RES*MC1
   + /D*E*/RES*MC1
   + /A*E*/RES*MC1
   + /B*E*/RES*MC1
   + /C*E*/RES*MC1

F := /A*F*/RES*MC1
   + /C*F*/RES*MC1
   + /B*F*/RES*MC1
   + /D*F*/RES*MC1
   + A*B*C*D*E*/F*/RES*MC1
   + /E*F*/RES*MC1

/HLD = /C + /D + /E + /F

FALE = HLDA*CLK*RASI*/MRASI

; SIMULATION NOT INCLUDED
```

Figure 7. Source Listing for the 8088 to Am2968 Interface (AmPAL16R6)

MC68000 to Am2968 Interface

Introduction

This application note shows how the general-purpose timing generator can be configured to interface between MC68000 and the Am2968. Figure 1 shows the block diagram of the interface. The correct interface signals must be adapted from the processor to the TimGen (PAL16L8) and the Am2968. Note, from Figure 2, that $\overline{AS}$ (Address Strobe) replaces $\overline{CYCREQ}$, and $\overline{UDS}$ (Upper Data Strobe) and LDS (Lower Data Strobe) replace $\overline{DS}$ and B/$\overline{W}$.

Asserting the Address Strobe signal ($\overline{AS}$), which is connected to the LE input, latches the valid address into the Am2968. $\overline{UDS}$ and $\overline{LDS}$ are controls indicating the flow of data on the data bus (D00–D15) during memory read/write operations. By asserting the correct polarity on these two signals ($\overline{UDS}$ & $\overline{LDS}$), either a byte or a word may be accessed.

The following sections will show the timing requirements which are being considered—that is, the desired processor operating frequency and the appropriate DRAM. Timing diagrams are also included to show the status of the interface signals during read/write operations for various processor frequencies.

MC68000 Timing Requirements

In generating the hardware for the timing interface for the MC68000 to the Am2968, the overall system timing requirements must be considered to guarantee that the desired memory access time can be met.

The following general equation formulates the parameters which must be considered in generating the read cycle time for the 68000 processor. The write cycle time may be similarly generated.

$$t_{READ} = 2t_{CYC} + t_{CH} - t_{CHSLx} - t_{DICL} - 2t_{PAL} - t_2 - t_{LATCH} + Nt_{CYC}$$

t_{READ} = read cycle time

t_{CYC} = clock period of the processor

t_{CH} = clock width High of the processor

t_{CHSLx} = clock High to $\overline{AS}$, $\overline{DS}$ Low of the processor

t_{DICL} = data in to clock Low (setup time) of the processor

t_{PAL} = programmable logic access time

t_2 = delay from RASI to $\overline{RASn}$ of the DMC

t_{LATCH} = delay through latch

N = number of inserted Wait States.

For simplicity, zero Wait States have been assumed in selecting the appropriate DRAMs. If Wait States had been asserted, the appropriate DRAM must be selected to meet the overall t_{READ} cycle time. Note that the access time of the DRAM (t_{ACC}) must be less than or equal to t_{READ} cycle time or else access to memory will be missed. The following evaluations show the calculated t_{ACC} based on the various processor operating frequencies: 8 MHz, 10 MHz and 12.5 MHz.

MC68000 W/WAIT STATES

8 MHz

$$t_{READ} = (250 + 55 - 60 - 15 - 30 - 23 - 13 + N * 125)\ ns = (164 + 125N)\ ns$$

DRAM t_{ACC} = 150 ns

10 MHz

$$t_{READ} = (200 + 45 - 55 - 10 - 30 - 23 - 13 + N * 100)\ ns = (114 + 100N)\ ns$$

DRAM t_{ACC} = 100 ns

12.5 MHz

$$t_{READ} = (160 + 35 - 55 - 10 - 30 - 23 - 13 + N * 80)\ ns = (64 + 80N)\ ns$$

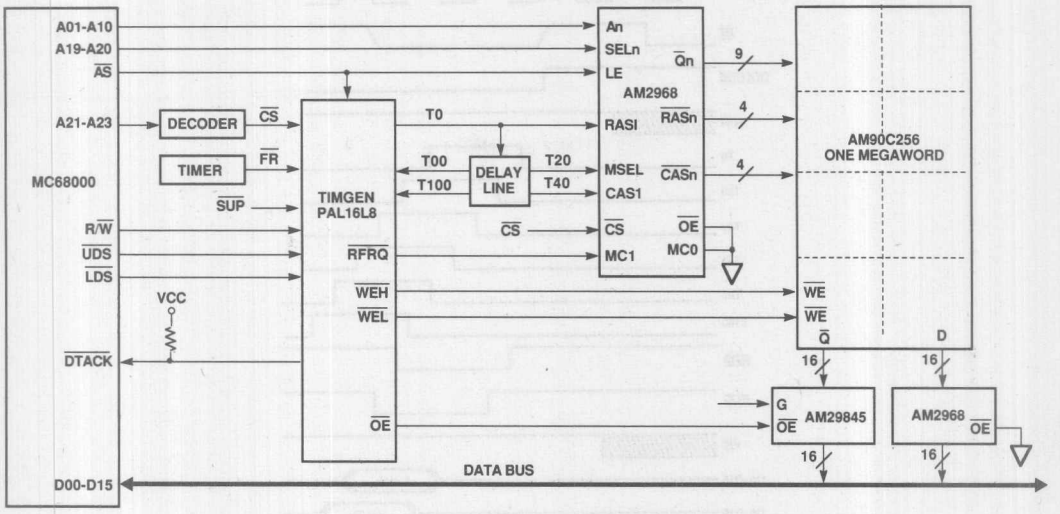


Figure 1. 68000 System Diagram

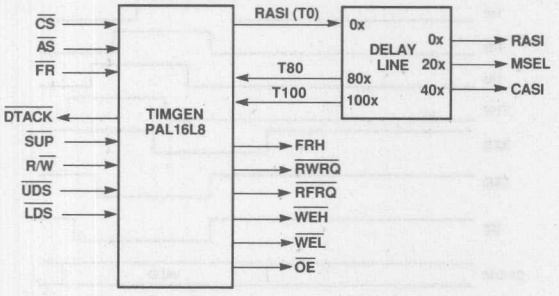


Figure 2. Interface Diagram

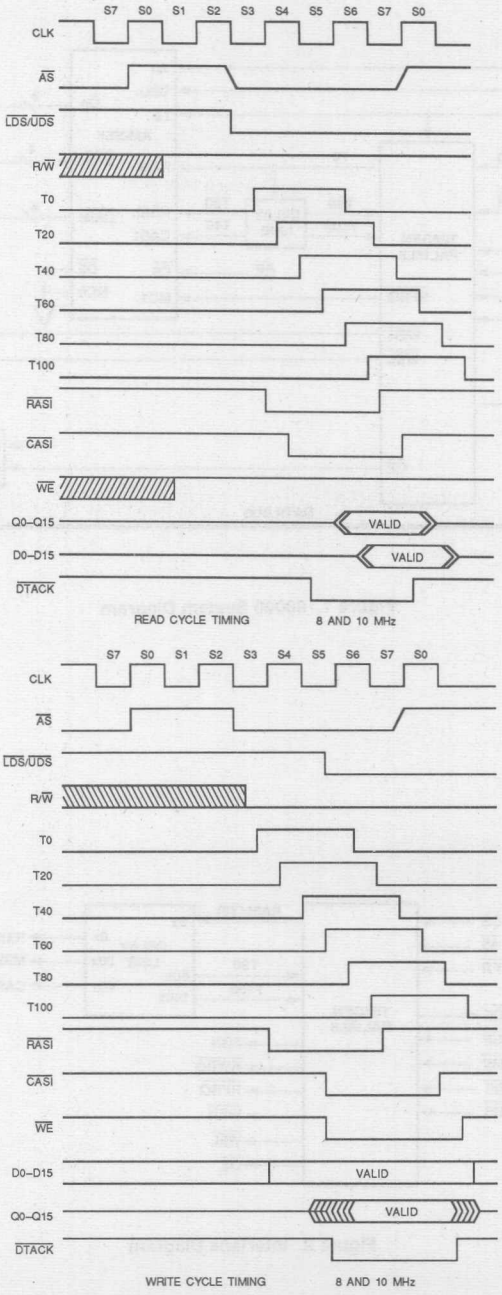
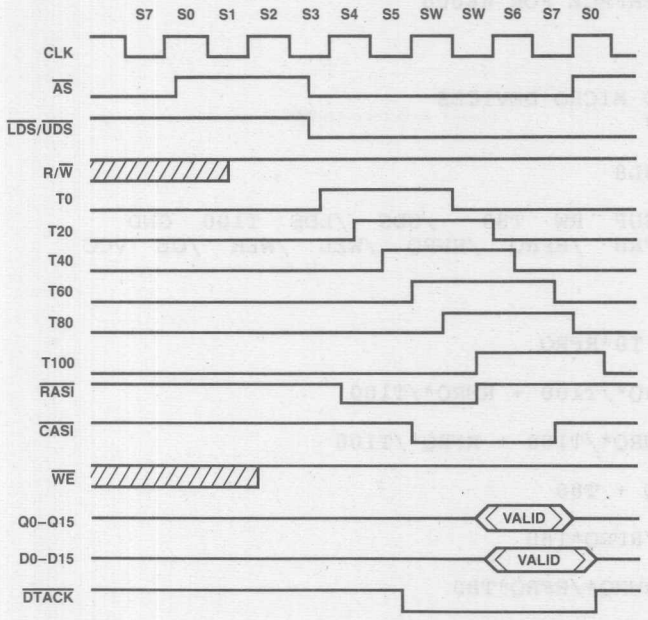


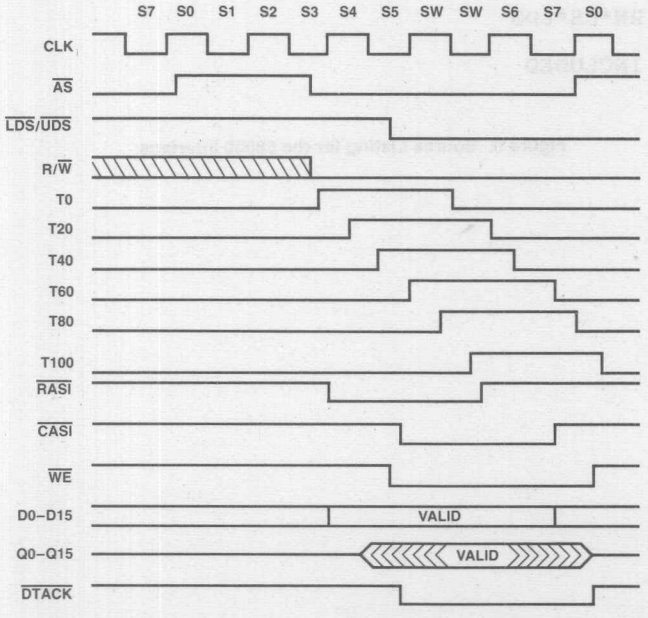
Figure 3. Read and Write Cycle Timing Diagram

For 12.5 MHz frequency, asserting at least one Wait State will allow t_{READ} = 144 ns, thus DRAM t_{ACC} = 120 ns.

Figure 3 shows the read and write cycle timing diagrams relative to the processor's cycle time. The 8 MHz and 10 MHz are shown on the same diagram because of their similarity. Figure 4 shows the read and write timing for the processor operating at 12.5 MHz with Wait States asserted during these operations.



READ CYCLE TIMING
12.5 MHz



WRITE CYCLE TIMING
12.5 MHz

Figure 4. Read and Write Cycle Timing Diagram

```
TITLE     GEN INTERFACE FOR 68000
PATTERN   TIMGEN
REVISION  01
AUTHOR    LEE/YEE
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP TIMGEN PAL16L8

/CS   /AS   /FR   /SUP   RW   T80    /UDS   /LDS   T100   GND
NC   /DTACK   T0   FRH   /RFRQ   /RWRQ   /WEL   /WEH   /OE   VCC

EQUATIONS

/FRH = FR*/FRH + T0*RFRQ

RWRQ = CS*AS*/RWRQ*/T100 + RWRQ*/T100

RFRQ = FR*FRH*/RWRQ*/T100 + RFRQ*/T100

/T0 = /RWRQ*/RFRQ + T80

DTACK = CS*RWRQ*/RFRQ*T80

DTACK.TRST = CS*RWRQ*/RFRQ*T80

WEH = /SUP*RWRQ*CS*/UDS*/RW

WEL = /SUP*RWRQ*CS*/LDS*/RW

OE = CS*RW*UDS + RW*CS*LDS

; SIMULATION NOT INCLUDED
```

Figure 5. Source Listing for the 68000 Interface

General-Purpose Dual-Port Arbiter

Introduction

This application note shows the implementation of a general-purpose dual-port arbiter interfacing three processors, a Dynamic Memory Controller (Am2968A DMC), and dynamic memory. The arbiter, implemented with two programmable array logic PAL devices, and a delay line, can provide a simple and complete arbitration solution operating in asynchronous mode.

The interface consists of two PAL devices (TIMGEN2A and TIMGEN2B), a timer, a delay line, and two multiplexers. Figure 1 shows the block diagram of the system which includes the arbiter logic implemented with the PAL devices. These two devices, TIMGEN2A and TIMGEN2B, provide the control signals to the DMC as well as to the dynamic memory. The purpose of the arbiter is to mediate when two processors request for memory cycles, or when refresh cycles need to be performed. The first PAL device, TIMGEN2A, generates the signals for the DMC and the Address MUX; it also contains the arbitration logic. The second device, TIMGEN2B, generates the control signals for the memory and the data bus latches.

The Am2968A Dynamic Memory Controller (DMC) interfaces between the processors and the dynamic memory array. The DMC provides the required addresses either for the memory or the refresh cycles. For memory cycles, the addresses are generated by the processor and latched into the DMC. For refresh cycles, the addresses are generated by a 20-bit counter internal to the DMC. When the DMC receives the correct control signals, it generates the appropriate address (Qn) and the Row and Column Address Strobes ($\overline{RAS}$n and $\overline{CAS}$n) that are required to perform memory read/write and refresh.

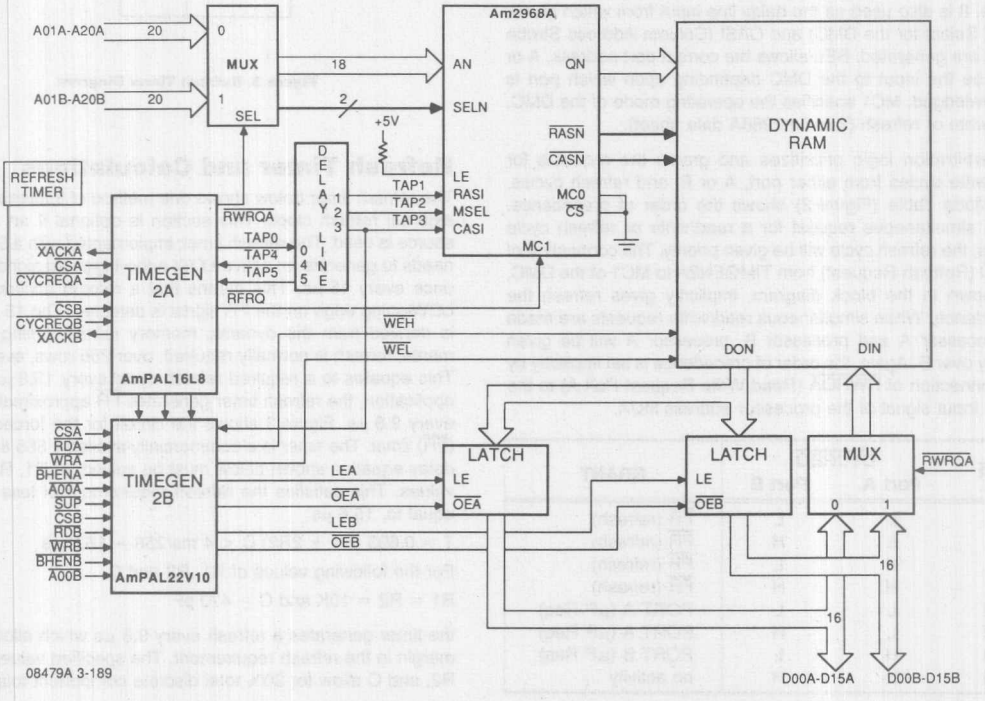


Figure 1. Block Diagram

Interface Overview

The dual-port arbiter can be interfaced to various processors, using the proper control signals from the particular processor. Typically, the processor interface signals are $\overline{CS}$ (Chip Select), $\overline{CYCREQ}$ (Cycle Request), RD (Read), and $\overline{WR}$ (Write), see Figure 1.

The two PAL devices, TIMGEN2A and TIMGEN2B are designed to operate in the following modes:

(1) generate proper control signals when cycle request is made
(2) arbitrate between two processor requests
(3) arbitrate between a refresh request and processor request
(4) arbitrate between a refresh request and two cycle requests

Requests by the processor are made when $\overline{CS}$ and $\overline{CYCREQ}$ are valid and the memory address is present. During dual-port arbitration, the occurrence of simultaneous requests, for instance, $\overline{CSA}$ (Chip Select Port A), $\overline{CYCREQA}$ (Cycle Request Port A) and $\overline{FR}$ (Forced Refresh), will result in the granting of a refresh cycle and the appropriate address being generated accordingly by the DMC.

TIMGEN2A generates the appropriate control signals to the data and address multiplexers, the delay line, and the memory controller. These control signals are: RASI (Row Address Strobe Input), SEL (Processor Address MUX Select), and MC1 (Mode Control Input). RASI is generated for all read/write and refresh cycles. It is also used as the delay line input from which MSEL (MUX Select for the DMC) and CASI (Column Address Strobe Input) are generated. SEL allows the correct port address, A or B, to be the input to the DMC depending upon which port is acknowledged. MC1 specifies the operating mode of the DMC, read/write or refresh (see Am2968A data sheet).

The arbitration logic prioritizes and grants the requests for read/write cycles from either port, A or B, and refresh cycles. The Mode Table (Figure 2) shows the order of precedence. When simultaneous request for a read/write or refresh cycle occurs, the refresh cycle will be given priority. The connection of $\overline{RFRQ}$ (Refresh Request) from TIMGEN2A to MC1 of the DMC, as shown in the block diagram, implicitly gives refresh the precedence. When simultaneous read/write requests are made by processor A and processor B, processor A will be given priority over B. Again, the order of precedence is set implicitly by the connection of $\overline{RWRQA}$ (Read/Write Request Port A) to the select input signal of the processor address MUX.

$\overline{FR}$	$\overline{CYCREQ}$ Port A	$\overline{CYCREQ}$ Port B	GRANT
L	L	L	$\overline{FR}$ (refresh)
L	L	H	$\overline{FR}$ (refresh)
L	H	L	$\overline{FR}$ (refresh)
L	H	H	$\overline{FR}$ (refresh)
H	L	L	PORT A (μP Req)
H	L	H	PORT A (μP Req)
H	H	L	PORT B (μP Req)
H	H	H	no activity

Figure 2. Mode Select Table

The second programmable logic device, TIMGEN2B generates the $\overline{OE}$ (Output Enable) signal for the data bus latches and the $\overline{WE}$ (Write Enable) signal for memory. The inputs to TIMGEN2B are the control signals—$\overline{CSA}$, $\overline{CSB}$, $\overline{RDA}$ (Read A), $\overline{RDB}$ (Read B), $\overline{WRA}$ (Write A), and $\overline{WRB}$ (Write B). Signals such as $\overline{BHEN}$ (Byte High Enable) and A00, the least significant bit of the address, have been included to show their function (if available). These two signals are provided by the processor and together they determine whether a word or byte transfer is to be performed during memory operations.

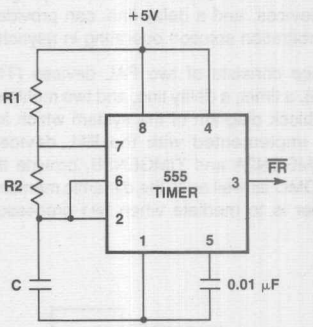


Figure 3. Refresh Timer Diagram

Refresh Timer and Calculations

The refresh timer below shows one method of implementing an external refresh clock. This section is optional if an alternate source is used. The refresh timer, implemented with a 555 timer, needs to generate an active-LOW edge-triggered signal at least once every 15 μs. This means that a refresh occurs when a LOW-going edge on the $\overline{FR}$ signal is detected. The 15 μs value is derived from the dynamic memory refresh timing requirements; refresh is normally required, over 256 rows, every 4 ms. This equates to a required refresh cycle every 15.6 μs. In this application, the refresh timer generates $\overline{FR}$ approximately once every 9.8 μs. Figure 3 shows the circuit for the forced refresh ($\overline{FR}$) timer. The timer is a commercially-available 555 timer. The delay equation shown below must be solved for R1, R2, and C values. This satisfies the refresh requirement of less than, or equal to, 15.6 μs

$T = 0.693\ (R1 + 2R2)\ C < 4\ ms/256 = 15.6\ \mu s$

For the following values of R1, R2 and C

$R1 = R2 = 10K$ and $C = 470\ pF$

the timer generates a refresh every 9.8 μs which allows some margin in the refresh requirement. The specified values for R1, R2, and C allow for 30% total discrete component tolerance.

TIMGEN2A

The function of TIMGEN2A is to generate $\overline{RASI}$ (Row Address Strobe Input) and arbitrate between processor and refresh cycles. Based on the arbitration, TIMGEN2A also generates $\overline{RWRQ}$ (Read/Write Request), or $\overline{RFRQ}$ (Refresh Request), that determines the type of cycle to be performed. TIMGEN2A can arbitrate between refresh and processor requests and also arbitrate between Port A and Port B processor requests.

When simultaneous refresh and processor requests are generated, refresh ($\overline{RFRQ}$) will be given priority. The priority is implicit by design because the $\overline{RFRQ}$ output of TIMGEN2A is connected to MC1 of the Am2968A DMC. When $\overline{RFRQ}$ becomes active (LOW), MC1 forces the DMC into refresh mode.

When simultaneous processor requests are made, $\overline{RWRQA}$ (Read/Write Request of Port A) will be granted before $\overline{RWRQB}$ (Read/Write Request of Port B). This priority is also implicit by design because the $\overline{RWRQA}$ output is connected to SEL of the processor's address multiplexer. When $\overline{RWRQA}$ becomes active, address from Port A will be selected for input to the Row and Column Address latches of the Am2968A, also, the data from Data Bus A will be allowed to flow to the inputs of the DRAMs. In either case, RASI must be generated to initiate any of the cycles. Once RASI is generated, the control signals, MSEL and CASI, will be generated from the delay line relative to RASI.

TIMGEN2B

The function of TIMGEN2B is to generate $\overline{WE}$ (Write Enable) for memory and also $\overline{OE}$ (Output Enable) to control the flow of data through the data bus latches. The inputs to TIMGEN2B are, $\overline{CSA}$, $\overline{RDA}$ (Read Strobe A), $\overline{WRA}$ (Write Strobe A), $\overline{CSB}$, $\overline{RDB}$ (Read Strobe B), and $\overline{WRB}$ (Write Strobe B). In addition, there are $\overline{BHENA}$ (Byte High Enable A) and $\overline{BHENB}$ (Byte High Enable B), A00A (least significant address bit for Port A) and A00B (least significant address bit for Port B). These signals are generated by the processor and are used to specify either byte or word transfers.

The following description outlines the operation of TIMGEN2B. If $\overline{CSA}$ and $\overline{RDA}$ become active (LOW), TIMGEN2B will cause $\overline{OEA}$ to become active (LOW), enabling the data bus latches and allowing data corresponding to address A to flow onto the system data bus (read operation). The identical procedure occurs for Port B when the control inputs for Port B is active. The block diagram shown in Figure 1 shows a multiplexer for the system data bus. This multiplexer controls the data flow into memory from either processor A or processor B, during write operations. The select to the MUX is controlled by $\overline{RWRQA}$, the same signal that controls the select to the processor address MUX, allowing address Port A to implicitly have higher priority during process memory access.

The following sections provide general information about interfacing the arbiter to various major types of processors; they are: the iAPX-type, the AmZ8000, and the MC68000-type processors. The PAL devices for the three types of processors will be identified as follows: TIMGEN2A and TIMGEN2B are for the iAPX-type; TIMGEN3A, TIMGEN3B for the AmZ8000; and TIMGEN4A and TIMGEN4B are for the MC68000-type. When mixing different processor in an interface, some of the control signals mentioned for the particular groups may need to be modified to be applicable to the processor under consideration. The designer can tailor the design to meet specific needs with simple modification to the PAL device logic equations provided.

Interfacing the iAPX-Type Processors

The control signals required for TIMGEN2A are shown in Figure 4. These signals are: $\overline{CS}$, $\overline{CYCREQ}$, $\overline{FR}$ and the delay line outputs (TAP1–TAP5). $\overline{CS}$ and $\overline{CYCREQ}$ are provided by the processor. $\overline{FR}$ is the refresh request from the refresh timer. Sources to this input can be provided either by the timer described in this application note or other appropriate sources. TAP1–TAP5 are the timing delay outputs used to regulate many of the signal generation. The outputs generated from this PAL device are TAP0 (RASI), $\overline{XACKA}$, $\overline{XACKB}$, and the grants for processor and refresh requests.

The signals for TIMGEN2B are $\overline{RD}$, $\overline{WR}$, $\overline{BHE}$, and A00. For byte/word transfers, the designer should consult the appropriate processor data sheet. In general, this processor group uses $\overline{BHE}$ in conjunction with A00 to define the transfer function of the data bus. The outputs from this PAL device are: Write Enables, Latch Enables, and Output Enables.

Interfacing the AmZ8000

The control signals required for this interface are shown in Figure 5. For TIMGEN3A, the general controls are $\overline{CS}$ and $\overline{CYCREQ}$, $\overline{FR}$ and the delay line outputs (TAP1–TAP5). $\overline{CS}$ and $\overline{CYCREQ}$ are generated by the processor. $\overline{FR}$ is the refresh request from the refresh timer. TAP1–TAP5 are the delay line outputs used to regulate the signal generation. The outputs from this PAL device are $\overline{WAITA}$, $\overline{WAITB}$, TAP0 (RASI), and the grants for processor and refresh requests.

The control signals for TIMGEN3B are R/$\overline{W}$, $\overline{DS}$, A00 and B/$\overline{W}$. For byte/word transfers, the designer should consult the processor data sheet for correct signal generation. The outputs from this PAL device are: Write Enables, Latch Enables and Output Enables.

Interfacing the MC68000-Type Processors

The control signals required for this interface are shown in Figure 6. For TIMGEN4A, the general controls are $\overline{CS}$ and $\overline{CYCREQ}$, $\overline{FR}$ and the delay line outputs (TAP1–TAP5). $\overline{CS}$ and $\overline{CYCREQ}$ are generated by the processor. $\overline{FR}$ is the refresh request from the refresh timer. TAP1–TAP5 are the delay line outputs used to regulate signal generation. The outputs generated from this PAL device are: $\overline{DTACKA}$, $\overline{DTACKB}$, TAP0 (RASI) and the grants for processor and refresh request.

The control signals for TIMGEN4B are R/$\overline{W}$, $\overline{UDS}$, and $\overline{LDS}$. For byte/word transfers, the designer should consult the respective processor data sheet for correct signal generation. For example, to obtain either byte or word transfers with the MC68000, three signals, $\overline{UDS}$, $\overline{LDS}$ and R/$\overline{W}$, must provide the correct levels to guarantee the correct data transfers, see MC68000 Data Bus Control Table in the User's Manual. The outputs generated from this PAL device are: Write Enables, Latch Enables, and Output Enables.

The programmable logic device equations for all three interfaces are shown in Figures 7, 8, and 9.

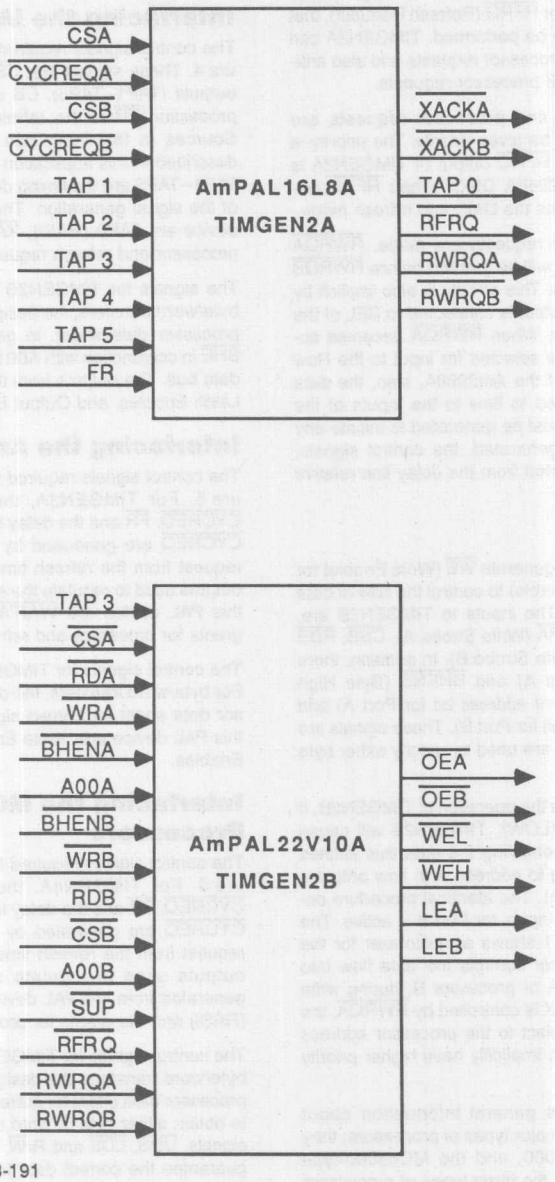
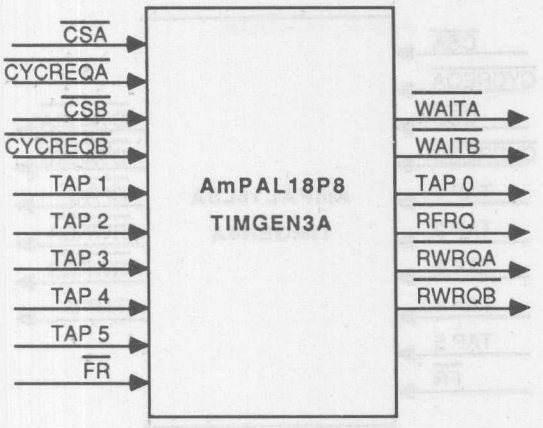
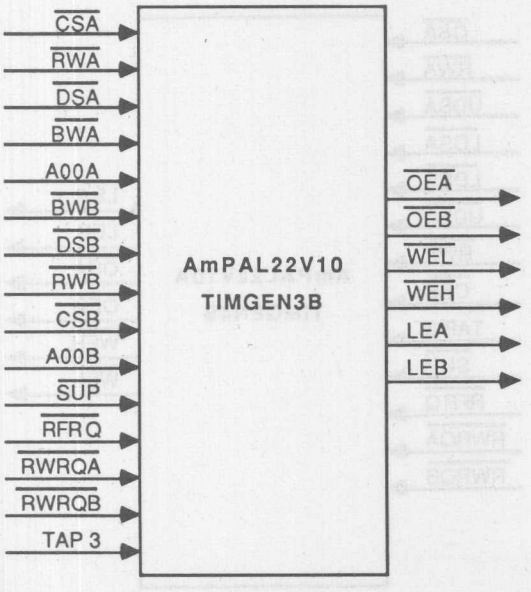


Figure 4. Interface for iAPX-Type Processors





08479A 3-192

Figure 5. Interface for Z8000-Type Processors

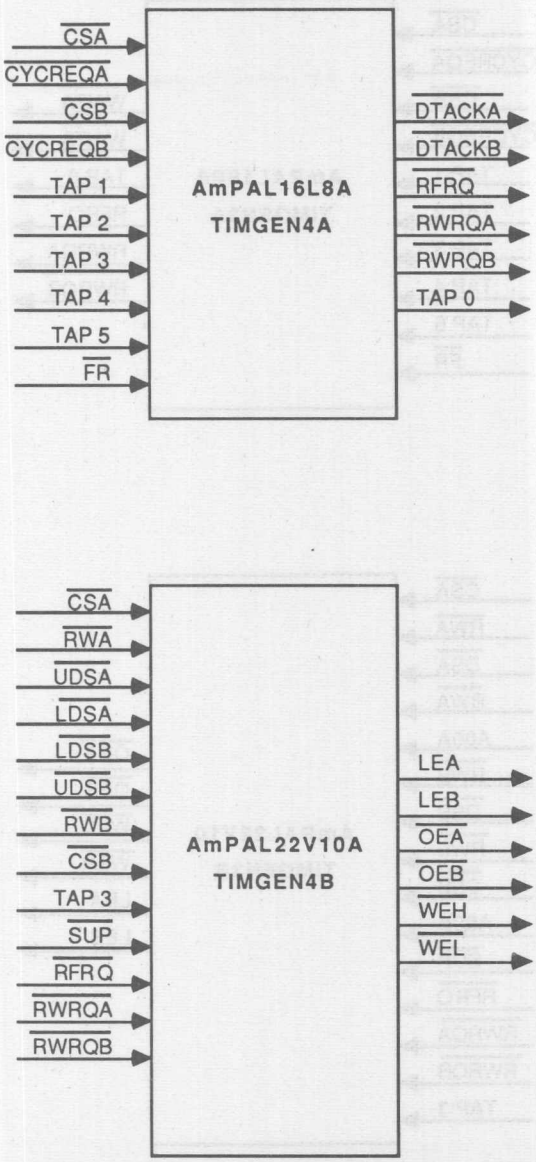


Figure 6. Interface for MC68000-Type Processors

```
TITLE    INTERFACE TIMING PAL DEVICE #1
PATTERN  A3-249A
REVISION 01
AUTHOR   J. ENGINEER
COMPANY  ADVANCED MICRO DEVICES
DATE     12/15/87

CHIP TIMGEN2A PAL16L8

/CSA /CYREQA /CSB  /CYREQB TAP1 TAP2  TAP3   TAP4  TAP5 GND
/FR  /XACKA  /XACKB TAP0   FRH /RFRQ /RWRQA /RWRQB NC   VCC

EQUATIONS

/FRH = FR*/FRH + TAP0*RFRQ

RFRQ = FR*FRH*/RWRQA*/RWRQB*/TAP5 + RFRQ*/TAP5

RWRQA = CSA*CYREQA*/RFRQ*/RWRQB*/TAP5 + RWRQA*/TAP5

RWRQB = CSB*CYREQB*/RFRQ*/RWRQA*/TAP5 + RWRQB*/TAP5

/TAP0 = /RFRQ*/RWRQA*/RWRQB + TAP4 ;PAL device for 8086,80186
                                   ;and 80286 processors
XACKA = CSA*RWRQA*/RFRQ*TAP4
XACKA.TRST = CSA*RWRQA*/RFRQ*TAP4

XACKB = CSB*RWRQB*/RFRQ*/RWRQA*TAP4
XACKB.TRST = CSB*RWRQB*/RFRQ*/RWRQA*TAP4


TITLE    INTERFACE TIMING PAL DEVICE #2
PATTERN  A3-249B
REVISION 01
AUTHOR   J. ENGINEER
COMPANY  ADVANCED MICRO DEVICES
DATE     12/15/87

CHIP TIMGEN2B PAL22V10

/CSA  /RDA /WRA /BHENA A00A /BHENB /WRB   /RDB /CSB NC  TAP3 GND
A00B /OEA /OEB /SUP  /RFRQ /RWRQA /RWRQB /WEL /WEH LEB LEA  VCC
GLOBAL

EQUATIONS ;PAL DEVICE FOR 8086 PROCESSORS

WEH = /SUP*RWRQA*/RFRQ*CSA*WRA*BHENA
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*WRB*BHENB

WEL = /SUP*RWRQA*/RFRQ*CSA*WRA*/A00A
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*WRB*/A00B

OEA = CSA*RWRQA*/RFRQ*RDA

OEB = CSB*RWRQB*/RFRQ*RDB

LEA = CSA*RWRQA*/RFRQ*RDA*TAP3

LEB = CSB*RWRQB*/RFRQ*RDB*TAP3
```

Figure 7. PLPL Specification for the Example of Figure 4

```
TITLE    INTERFACE TIMING PAL DEVICE #1
PATTERN  A3-250A
REVISION 01
AUTHOR   J. ENGINEER
COMPANY  ADVANCED MICRO DEVICES
DATE     12/15/87

CHIP TIMGEN3A PAL18P8

/CSA /CYREQA /CSB  /CYREQB TAP1 TAP2  TAP3   TAP4  TAP5 GND
/FR  /WAITA  /WAITB TAP0   FRH /RFRQ /RWRQA /RWRQB NC   VCC

EQUATIONS

/FRH = FR*/FRH + TAP0*RFRQ

RFRQ = FR*FRH*/RWRQA*/RWRQB*/TAP5 + RFRQ*/TAP5

RWRQA = CSA*CYREQA*/RFRQ*/RWRQB*/TAP5 + RWRQA*/TAP5

RWRQB = CSB*CYREQB*/RFRQ*/RWRQA*/TAP5 + RWRQB*/TAP5

/TAP0 = /RFRQ*/RWRQA*/RWRQB + TAP4   ;PAL device for Z8000
                                     ;processor
/WAITA = CSA*RWRQA*/RFRQ*TAP4
WAITA.TRST = CSA*RWRQA*/RFRQ*TAP4

/WAITB = CSB*RWRQB*/RFRQ*/RWRQA*TAP4
WAITB.TRST = CSB*RWRQB*/RFRQ*/RWRQA*TAP4

TITLE    INTERFACE TIMING PAL DEVICE #2
PATTERN  A3-250B
REVISION 01
AUTHOR   J. ENGINEER
COMPANY  ADVANCED MICRO DEVICES
DATE     12/15/87

CHIP TIMGEN3B PAL22V10

/CSA   RWA /DSA  BWA  A00A  BWB   /DSB    RWB /CSB NC  TAP3 GND
A00B /OEA /OEB /SUP /RFRQ /RWRQA /RWRQB /WEL /WEH LEB LEA  VCC
GLOBAL

EQUATIONS ;PAL DEVICE FOR Z8000 PROCESSOR

WEH = /SUP*RWRQA*/RFRQ*CSA*/RWA*/BWA
    + /SUP*RWRQA*/RFRQ*CSA*/RWA*BWA*A00A
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*/RWB*/BWB
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*/RWB*BWB*A00B

WEL = /SUP*RWRQA*/RFRQ*CSA*/RWA*/BWA
    + /SUP*RWRQA*/RFRQ*CSA*/RWA*BWA*/A00A
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*/RWB*/BWB
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*/RWB*BWB*/A00B

OEA = CSA*RWRQA*/RFRQ*RWA*DSA

OEB = CSB*RWRQB*/RFRQ*RWB*DSB

LEA = CSA*RWRQA*/RFRQ*RWA*TAP3

LEB = CSB*RWRQB*/RFRQ*RWB*TAP3
```

Figure 8. PLPL Specification for the Example of Figure 5

```
TITLE     INTERFACE TIMING PAL DEVICE #1
PATTERN   A3-251A
REVISION  01
AUTHOR    J. ENGINEER
COMPANY   ADVANCED MICRO DEVICES
DATE      12/15/87

CHIP TIMGEN4A PAL16L8

/CSA /CYREQA /CSB  /CYREQB TAP1 TAP2  TAP3   TAP4  TAP5 GND
/FR  /DTACKA /DTACKB TAP0  FRH /RFRQ /RWRQA /RWRQB NC   VCC

EQUATIONS

/FRH = FR*/FRH + TAP0*RFRQ

RFRQ = FR*FRH*/RWRQA*/RWRQB*/TAP5 + RFRQ*/TAP5

RWRQA = CSA*CYREQA*/RFRQ*/RWRQB*/TAP5 + RWRQA*/TAP5

RWRQB = CSB*CYREQB*/RFRQ*/RWRQA*/TAP5 + RWRQB*/TAP5

/TAP0 = /RFRQ*/RWRQA*/RWRQB + TAP4   ;PAL device for 68000
                                     ;processor
DTACKA = CSA*RWRQA*/RFRQ*TAP4
DTACKA.TRST = CSA*RWRQA*/RFRQ*TAP4

DTACKB = CSB*RWRQB*/RFRQ*/RWRQA*TAP4
DTACKB.TRST = CSB*RWRQB*/RFRQ*/RWRQA*TAP4


TITLE     INTERFACE TIMING PAL DEVICE #2
PATTERN   A3-251B
REVISION  01
AUTHOR    J. ENGINEER
COMPANY   ADVANCED MICRO DEVICES
DATE      12/15/87

CHIP TIMGEN4B PAL22V10

/CSA   RWA /UDSA /LDSA NC5  /LDSB  /UDSB   RWB /CSB NC TAP3 GND
NC10 /OEA /OEB  /SUP /RFRQ /RWRQA /RWRQB /WEL /WEH LEB LEA VCC
GLOBAL

EQUATIONS ;PAL DEVICE FOR 68000 PROCESSOR

WEH = /SUP*RWRQA*/RFRQ*CSA*/RWA*UDSA
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*/RWB*UDSB

WEL = /SUP*RWRQA*/RFRQ*CSA*/RWA*LDSA
    + /SUP*RWRQB*/RFRQ*/RWRQA*CSB*/RWB*LDSB

OEA = CSA*RWRQA*/RFRQ*RWA*(UDSA + LDSA)

OEB = CSB*RWRQB*/RFRQ*RWB*(UDSB + LDSB)

LEA = CSA*RWRQA*/RFRQ*RWA*TAP3

LEB = CSB*RWRQB*/RFRQ*RWB*TAP3
```

Figure 9. PLPL Specification for the Example of Figure 6

Dynamic Memory Control State Sequencer

An example of a control path application for a PAL device is in a memory system. Most large memory systems use MOS dynamic RAMs. Their high density allows packing a large memory size into a small board area. Dynamic RAM prices also make them very cost effective.

Dynamic RAMs require external logic for address multiplexing, timing generation and refresh control. This application note shows the use of a PAL16R8A and an Am2964B to provide the necessary external logic for a typical dynamic memory system. The PAL device is used as a state sequencer for timing generation and the Am2964B provides specialized control circuitry and reduces timing skew between control signals. This implementation replaces about 20 SSI/MSI packages.

Design Requirements

A system block diagram is shown in Figure 1. The control bus provides most of the inputs to the PAL state sequencer. These include: Memory Request ($\overline{MREQ}$), READ/$\overline{WRITE}$ (R$\overline{W}$), $\overline{RESET}$ ($\overline{RST}$), Refresh Clock (RFCK), and Read-Modify-Write (RMW). Two upper address lines of the address bus serve as board selects (BS_1, BS_0), and one local signal, $\overline{SLOW}$/FAST Memory (FAST), allows use of either slow or fast memory. A READ/$\overline{WRITE}$ sequence is initialized by $\overline{MREQ}$ ANDed with the proper board select conditions and a refresh sequence is initialized by RFCK. If both sequences are requested at the same time, a refresh sequence is performed. R$\overline{W}$ when HIGH selects a READ operation and when LOW selects a WRITE operation. RMW when HIGH selects a Read-Modify-Write cycle.

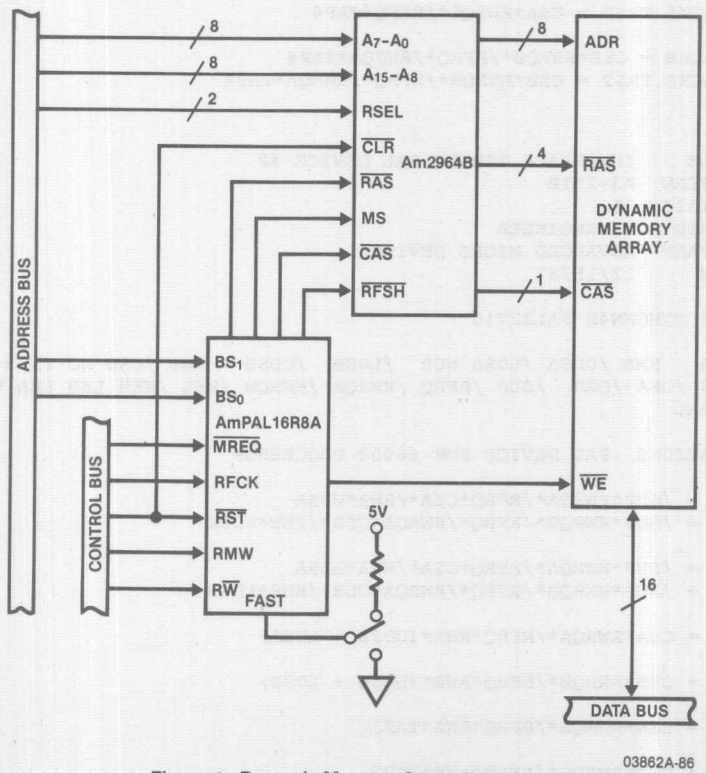


Figure 1. Dynamic Memory Controller

The outputs of the PAL device provide the timing and control inputs to the Am2964B. These are: Row Address Strobe ($\overline{RAS}$), Address Multiplexer Select (MS), Column Address Strobe ($\overline{CAS}$), and Refresh ($\overline{RFSH}$). In addition, the PAL device provides the Write Enable ($\overline{WE}$) to the Memory Array. Figure 1 shows the timing for fast READ/WRITE cycles. The memory cycle is initiated by $\overline{MREQ}$ going LOW. The PAL device responds by bringing $\overline{RAS}$ LOW at t_0, followed by MS going LOW at t_1, and finally bringing $\overline{CAS}$ LOW at t_2. If RW is LOW, $\overline{WE}$ is also brought LOW at t_2. $\overline{WE}$ is held LOW until t_4. $\overline{RAS}$, MS and $\overline{CAS}$ are brought HIGH at t_5. The rising edge of any of these three signals may be used to latch output data during a Read operation. The state sequencer then disabled for three states to allow for memory precharge.

By holding the FAST input LOW, an extended memory cycle is available to accommodate slower RAMs. The timing appears in Figure 3.

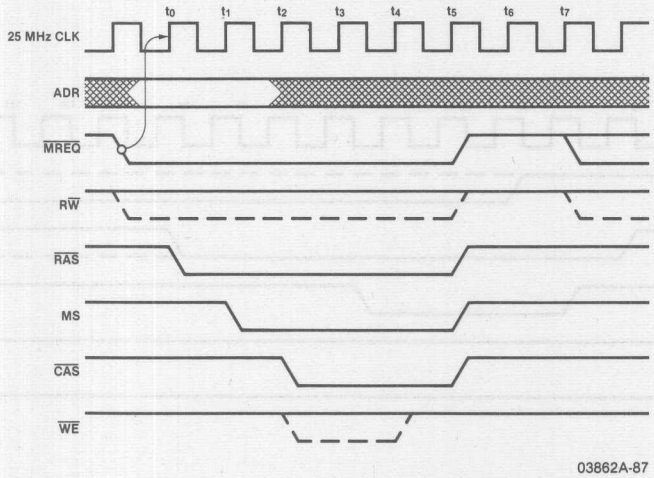


Figure 2. Fast READ/WRITE Cycle

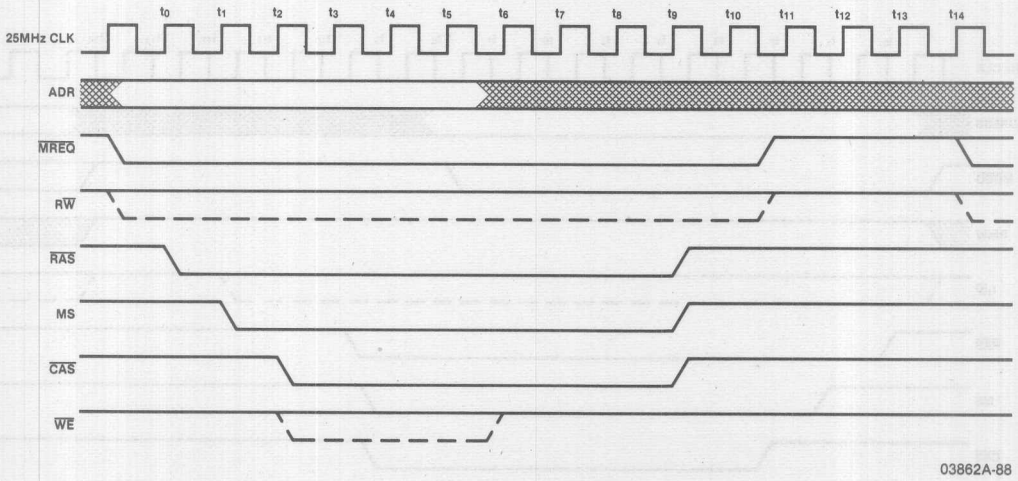


Figure 3. Extended Memory Cycle

$\overline{RAS}$-Only refresh cycle timing is shown in Figure 4. The refresh cycle is initiated when RFCK goes HIGH. The $\overline{RFSH}$ output goes LOW at t_0, followed by $\overline{RAS}$ at t_1. The Am2964B supplied the necessary refresh address. $\overline{RAS}$ is brought back HIGH at t_5 and precharge is then timed out. An extended refresh cycle for slower memory is available also. Burst refresh can be accomplished by leaving RFCK HIGH for as many refresh cycles as desired.

Read-Modify-Write cycle timing is activated by setting RMW HIGH. This is especially valuable in systems with Error Detection/Correction (EDC) capability. Data can be read, modified by the EDC circuitry (Am2960), and if necessary, written back to memory in a single memory cycle. Read-Modify-Write cycle timing is shown in Figure 5. Note that $\overline{WE}$ goes LOW at the end of the cycle.

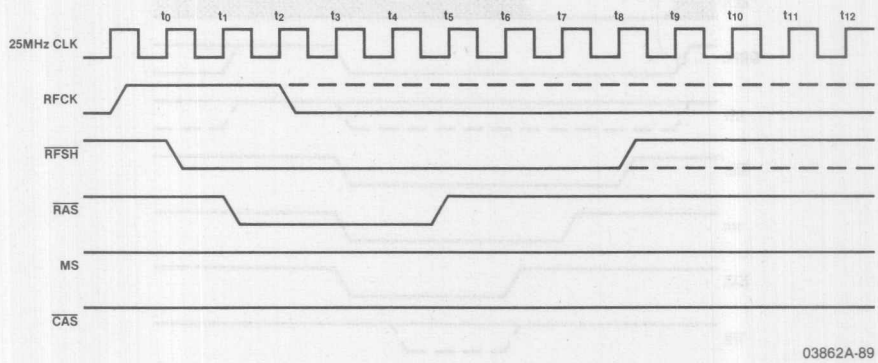


Figure 4. $\overline{RAS}$-Only Refresh Cycle

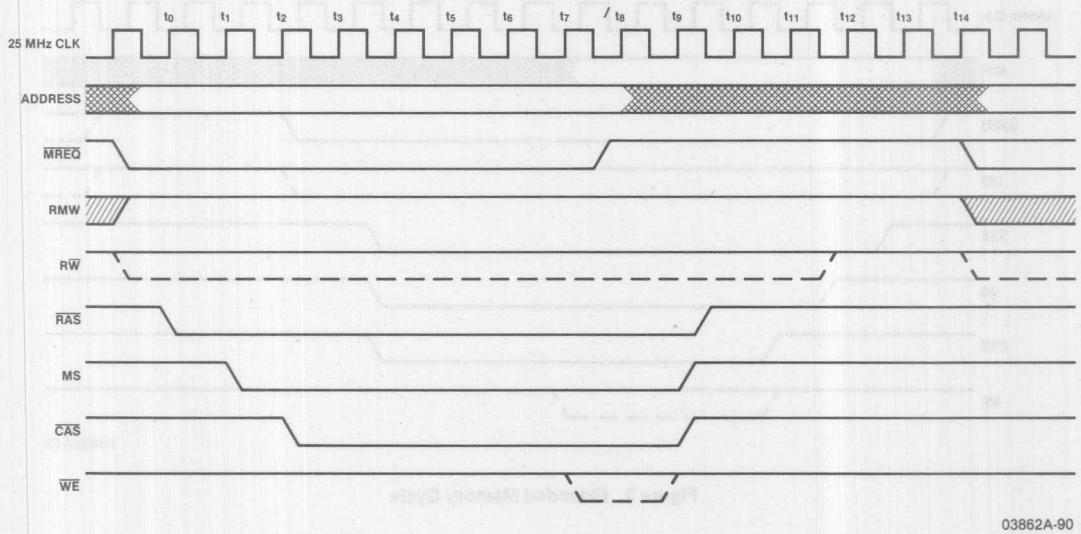


Figure 5. Read-Modify-Write Cycle

Design Approach

The first step in the state sequencer design process is to define the timing waveforms for all of the functions desired. Figures 2, 3, 4, and 5 are the result. Next, characteristics of the resulting waveforms are examined. Initially, the sequencer is waiting on the $\overline{MREQ}$ or RFCK input. If $\overline{MREQ}$ goes LOW, the $\overline{RAS}$ to MS to $\overline{CAS}$ sequence is initiated. If RFCK goes HIGH, the $\overline{RFSH}$ to $\overline{RAS}$ sequence is initiated. Both sequences are equivalent to a simple "shift" function. Once the shift sequence is completed and the signals are asserted, they must stay asserted for a specific time depending on the selected function. To time the length that signals must stay asserted requires a "counting" function. The precharge sequence at the end of all cycles also requires "counting". This partitions most of the design into two smaller functional blocks; a shifter and a counter. The remaining function select and control logic is partitioned into a "multiplexer-like" functional block. Figure 6 shows the PAL device partitioned into functional blocks. By dividing the design into blocks, its implementation becomes simple.

Figure 7 shows PAL device equations for the PAL16R8A dynamic memory state sequencer.

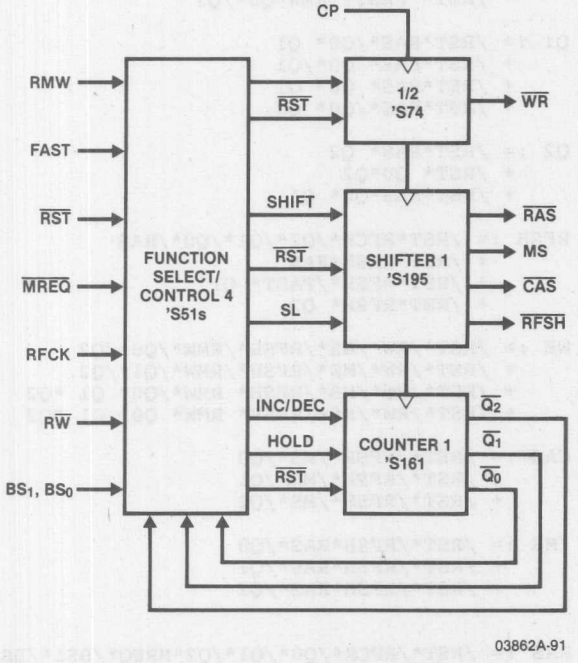


Figure 6. Partitioned Design/PAL Device Equivalent

```
TITLE    DYNAMIC MEMORY STATE SEQUENCER
PATTERN  AMP3-285
REVISION 01
NAME     JOE ENGINEER
COMPANY  ADVANCED MICRO DEVICES
DATE     11/25/87

CHIP MEMORY1 PAL16R8

CK  RFCK /RST RW /MREQ  RMW FAST BS1 BS0 GND
/E /Q0   /Q1 /Q2 /RFSH /WE /CAS  MS /RAS VCC

EQUATIONS

Q0 := /RST*/MS*/Q0
    + /RST*RFSH*RAS*/Q0
    + /RST*/FAST*/Q0*Q2
    + /RST*/FAST*/Q0*Q1
    + /RST*/FAST* Q1*Q2
    + /RST* FAST*/RMW*Q0*/Q1
    + /RST* FAST*/RMW*Q0*/Q2

Q1 := /RST*RAS*/Q0* Q1
    + /RST*RAS* Q0*/Q1
    + /RST*RAS* Q0* Q1
    + /RST*RAS*/Q0* Q2

Q2 := /RST*RAS* Q2
    + /RST* Q0*Q2
    + /RST*RAS*Q0* Q1

RFSH := /RST*RFCK*/Q2*/Q1*/Q0*/RAS
      + /RST*RFSH*RAS
      + /RST*RFSH*/FAST* Q1
      + /RST*RFSH* Q2

WE := /RST*/RW*/MS*/RFSH*/RMW*/Q0*/Q2
    + /RST*/RW*/MS*/RFSH*/RMW*/Q1*/Q2
    + /RST*/RW*/MS*/RFSH* RMW*/Q0* Q1 *Q2
    + /RST*/RW*/MS*/RFSH* RMW* Q0*/Q1 *Q2

CAS := /RST*/RFSH*/MS*/Q0
     + /RST*/RFSH*/MS*/Q1
     + /RST*/RFSH*/MS*/Q2

/MS := /RST*/RFSH*RAS*/Q0
     + /RST*/RFSH*RAS*/Q1
     + /RST*/RFSH*RAS*/Q2


RAS := /RST*/RFCK*/Q0*/Q1*/Q2*MREQ*/BS1*/BS0
     + /RST*/RFSH*/Q0*/Q1*/Q2*MREQ*/BS1*/BS0
     + /RST* RFSH*/Q0*/Q1*/Q2
     + /RST*RAS*/Q0
     + /RST*RAS*/Q1
     + /RST*RAS*/Q2
```

Figure 7. Source Listing for Dynamic Memory Control State Sequencer

8-Bit Error Detection and Correction

Single bit error detection and correction for an 8-bit data word requires 4 check bits, making a 12-bit code word. The simplest code to design is a 12-bit Hamming code. To arrive at the code, we set up the following matrix:

	B7	B6	B5	B4	B3	B2	B1	B0	C3	C2	C1	C0
S3	X	X	X	X					X			
S2	X				X	X	X			X		
S1		X	X		X	X		X			X	
S0		X		X	X		X	X				X

The vertical columns are in a counting pattern, excluding the single bit values of 8, 4, 2, 1. The single bit values are assigned to the check bits C3-C0. By reading horizontally across the rows of the matrix, we get the check equations by exclusive OR'ing the data bits with X's in that row and equating that to the check bit with an X in that row:

$$C3 = B7 \oplus B6 \oplus B5 \oplus B4$$
$$C2 = B7 \oplus B3 \oplus B2 \oplus B1$$
$$C1 = B6 \oplus B5 \oplus B3 \oplus B2 \oplus B0$$
$$C0 = B6 \oplus B4 \oplus B3 \oplus B1 \oplus B0$$

The check bits are stored along with the data bits in the following message format:

M12	M11	M10	M9	M8	M7	M6	M5	M4	M3	M2	M1
B7	B6	B5	B4	C3	B3	B2	B1	C2	B0	C1	C0

When a read occurs, the check bits are recalculated and compared with the stored bits to generate the 4 bit syndrome:

$$S3 = B7 \oplus B6 \oplus B5 \oplus B4 \oplus C3$$
$$S2 = B7 \oplus B3 \oplus B2 \oplus B1 \oplus C2$$
$$S1 = B6 \oplus B5 \oplus B3 \oplus B2 \oplus B0 \oplus C1$$
$$S0 = B6 \oplus B4 \oplus B3 \oplus B1 \oplus B0 \oplus C0$$

The 4 syndrome bits indicate the location of any single bit errors in the 12-bit message format which may than be corrected by inversion.

The Hamming code works by introducing enough other code words to create a difference of exactly 3 bits between legal code words. All other code words are illegal. If, in storage, one bit flips, the result is an illegal word. In addition, there is only one word in the set of legal code words from which it could have come, hence the correction.

The Hamming matrix and resultant sets of check bit and syndrome equations are selected so that when a single bit error occurs, the syndrome gives the position of that bit (either data or check) in the 12-bit message format.

Example	**B7**							**B0**
random data word:	1	1	0	1	1	1	0	0

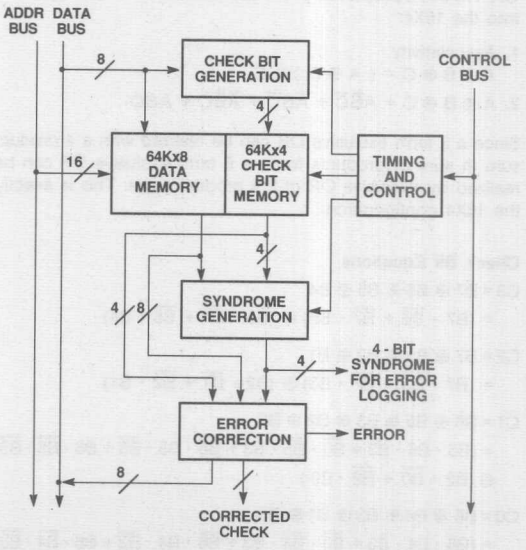
check bits:

$$C3 = B7 \oplus B6 \oplus B5 \oplus B4$$
$$= 1 \oplus 1 \oplus 0 \oplus 1 = 1$$
$$C2 = B7 \oplus B3 \oplus B2 \oplus B1$$
$$= 1 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C1 = B6 \oplus B5 \oplus B3 \oplus B2 \oplus B0$$
$$= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C0 = B6 \oplus B4 \oplus B3 \oplus B1 \oplus B0$$
$$= 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 1$$

message:

M12	M11	M10	M9	M8	M7	M6	M5	M4	M3	M2	M1
1	1	0	1	1	1	1	0	1	0	1	1
B7	B6	B5	B4	C3	B3	B2	B1	C2	B0	C1	C0

EDAC System Block Diagram



assume no error:

$S3 = B7 \oplus B6 \oplus B5 \oplus B4 \oplus C3$
$= 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 0$

$S2 = B7 \oplus B3 \oplus B2 \oplus B1 \oplus C2$
$= 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$

$S1 = B6 \oplus B5 \oplus B3 \oplus B2 \oplus B0 \oplus C1$
$= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$

$S0 = B6 \oplus B4 \oplus B3 \oplus B1 \oplus B0 \oplus C0$
$= 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 0$

syndrome 0000 indicates no error

assume data bit B7 flips (1 → 0):

$S3 = \underline{0} \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1$
$S2 = \underline{0} \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 1$
$S1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$
$S0 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 0$

syndrome 1100 indicates M12 or B7 is in error

assume check bit C3 flips (1 → 0):

$S3 = 1 \oplus 1 \oplus 0 \oplus 1 \oplus \underline{0} = 1$
$S2 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$
$S1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$
$S0 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 0$

syndrome 1000 indicates M8 or C3 is in error

While the check bits can be generated with a 256x4 PROM if the tolerances are loose, high performance systems will need to latch or register the check bits to meet cycle time requirements. Similarly, the syndrome bits can be generated with a 4096x4 PROM but in both cases the PAL16X4 is the high performance choice.

The worst case equations are S1 and S0 with a 6 term exclusive OR. We use 2 properties of the exclusive OR to fit the equations into the 16X4:

1. Associativity
   $A \oplus B \oplus C = (A \oplus B) \oplus C$
2. $A \oplus B \oplus C = A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + \overline{A}\,\overline{B}C + ABC$

Since a 3 term exclusive OR can be realized with a 4 product sum in sum of products form, a 6 term exclusive OR can be realized by exclusive OR of 2 4 product sums. This is exactly the 16X4 configuration.

Check Bit Equations

$C3 = B7 \oplus B6 \oplus B5 \oplus B4$
$= (B7 \cdot \overline{B6} + \overline{B7} \cdot B6) \oplus (B5 \cdot \overline{B4} + \overline{B5} \cdot B4)$

$C2 = B7 \oplus B3 \oplus B2 \oplus B1$
$= (B7 \cdot \overline{B3} + \overline{B7} \cdot B3) \oplus (B2 \cdot \overline{B1} + \overline{B2} \cdot B1)$

$C1 = B6 \oplus B5 \oplus B3 \oplus B2 \oplus B0$
$= (B6 \cdot B5 \cdot B3 + \overline{B6} \cdot \overline{B5} \cdot B3 + \overline{B6} \cdot B5 \cdot \overline{B3} + B6 \cdot \overline{B5} \cdot \overline{B3})$
$\oplus (B2 \cdot \overline{B0} + \overline{B2} \cdot B0)$

$C0 = B6 \oplus B4 \oplus B3 \oplus B1 \oplus B0$
$= (B6 \cdot B4 \cdot B3 + \overline{B6} \cdot \overline{B4} \cdot B3 + \overline{B6} \cdot B4 \cdot \overline{B3} + B6 \cdot \overline{B4} \cdot \overline{B3})$
$\oplus (B1 \cdot \overline{B0} + \overline{B1} \cdot B0)$

Syndrome Bit Equations

$S3 = B7 \oplus B6 \oplus B5 \oplus B4 \oplus C3$
$= (B7 \cdot B6 \cdot B5 + \overline{B7} \cdot \overline{B6} \cdot B5 + \overline{B7} \cdot B6 \cdot \overline{B5} + B7 \cdot \overline{B6} \cdot \overline{B5})$
$\oplus (B4 \cdot \overline{C3} + \overline{B4} \cdot C3)$

$S2 = B7 \oplus B3 \oplus B2 \oplus B1 \oplus C2$
$= (B7 \cdot B3 \cdot B2 + \overline{B7} \cdot \overline{B3} \cdot B2 + \overline{B7} \cdot B3 \cdot \overline{B2} + B7 \cdot \overline{B3} \cdot \overline{B2})$
$\oplus (B1 \cdot \overline{C2} + \overline{B1} \cdot C2)$

$S1 = B6 \oplus B5 \oplus B3 \oplus B2 \oplus B0 \oplus C1$
$= (B6 \cdot B5 \cdot B3 + \overline{B6} \cdot \overline{B5} \cdot B3 + \overline{B6} \cdot B5 \cdot \overline{B3} + B6 \cdot \overline{B5} \cdot \overline{B3})$
$\oplus (\overline{B2} \cdot \overline{B0} \cdot C1 + \overline{B2} \cdot B0 \cdot C1 + B2 \cdot B0 \cdot \overline{C1} + B2 \cdot \overline{B0})$

$S0 = B6 \oplus B4 \oplus B3 \oplus B1 \oplus B0 \oplus C0$
$= (B6 \cdot B4 \cdot B3 + \overline{B6} \cdot \overline{B4} \cdot B3 + \overline{B6} \cdot B4 \cdot \overline{B3} + B6 \cdot \overline{B4} \cdot \overline{B3})$
$\oplus (B1 \cdot B0 \cdot C0 + \overline{B1} \cdot \overline{B0} \cdot C0 + \overline{B1} \cdot B0 \cdot \overline{C0} + B1 \cdot \overline{B0} \cdot C0)$

The error correction block decodes the 4 syndrome bits, and, if they are not 0000, inverts the indicated bit in the message format. The equations:

$M12\ (= B7C) = S3 \cdot S2 \cdot \overline{S1} \cdot \overline{S0} \oplus B7$
$= S3 \cdot S2 \cdot \overline{S1} \cdot \overline{S0} \cdot \overline{B7} + \overline{(S3 \cdot S2 \cdot \overline{S1} \cdot \overline{S0})} \cdot B7$
$= S3 \cdot S2 \cdot \overline{S1} \cdot \overline{S0} \cdot \overline{B7} + \overline{S3} \cdot B7 + S2 \cdot B7 + S1 \cdot B7 + \overline{S0} \cdot B7$

$M11\ (= B6C) = S3 \cdot \overline{S2} \cdot S1 \cdot S0 \cdot \overline{B6} \cdot \overline{S3} \cdot B6 \cdot S2 \cdot B6 + \overline{S1} \cdot B6 \cdot \overline{S0} \cdot B6$

$M10\ (= B5C) = S3 \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot \overline{B5} + \overline{S3} \cdot B5 + S2 \cdot B5 + \overline{S1} \cdot B5 + S0 \cdot B5$

$M9\ (= B4C) = S3 \cdot \overline{S2} \cdot \overline{S1} \cdot S0 \cdot \overline{B4} + \overline{S3} \cdot B4 + S2 \cdot B4 + S1 \cdot B4 + \overline{S0} \cdot B4$

$M8\ (= C3C) = S3 \cdot \overline{S2} \cdot \overline{S1} \cdot \overline{S0} \cdot \overline{C3} + \overline{S3} \cdot C3 + S2 \cdot C3 + S1 \cdot C3 + S0 \cdot C3$

$M7\ (= B3C) = \overline{S3} \cdot S2 \cdot S1 \cdot S0 \cdot \overline{B3} + S3 \cdot B3 + \overline{S2} \cdot B3 + \overline{S1} \cdot B3 + \overline{S0} \cdot B3$

$M6\ (= B2C) = \overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0} \cdot \overline{B2} + S3 \cdot B2 + \overline{S2} \cdot B2 + \overline{S1} \cdot B2 + S0 + B2$

$M5\ (= B1C) = \overline{S3} \cdot S2 \cdot \overline{S1} \cdot S0 \cdot \overline{B1} + S3 \cdot B1 + \overline{S2} \cdot B1 + S1 \cdot B1 + \overline{S0} \cdot B1$

$M4\ (= C2C) = \overline{S3} \cdot S2 \cdot \overline{S1} \cdot \overline{S0} \cdot \overline{C2} + S3 \cdot C2 + \overline{S2} \cdot C2 + S1 \cdot C2 + S0 \cdot C2$

$M3\ (= B0C) = \overline{S3} \cdot \overline{S2} \cdot S1 \cdot S0 \cdot \overline{B0} + S3 \cdot B0 + S2 \cdot B0 + \overline{S1} \cdot B0 + \overline{S0} \cdot B0$

$M2\ (= C1C) = \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot \overline{C1} + S3 \cdot C1 + S2 \cdot C1 + \overline{S1} \cdot C1 + S0 \cdot C1$

$M1\ (= C0C) = \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot S0 \cdot \overline{C0} + S3 \cdot C0 + S2 \cdot C0 + S1 \cdot C0 + \overline{S0} \cdot C0$

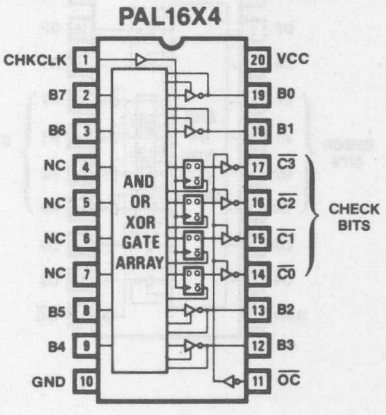
$ERROR = \overline{S3 \cdot S2 \cdot S1 \cdot S0}$

ERROR is an active high error indicator available for error logging along with the 4 syndrome bits.

To use 2 PAL16L8's for the error correction block, we need only invert the message bits to get active true outputs.

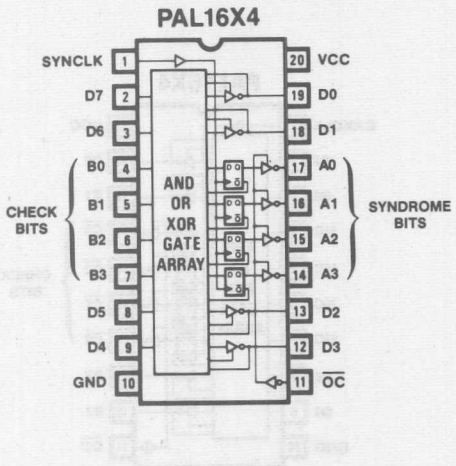
Description

This PAL device generates 4 check bits in a 12 bit hamming code word to provide error detection and correction on an 8 bit data word.



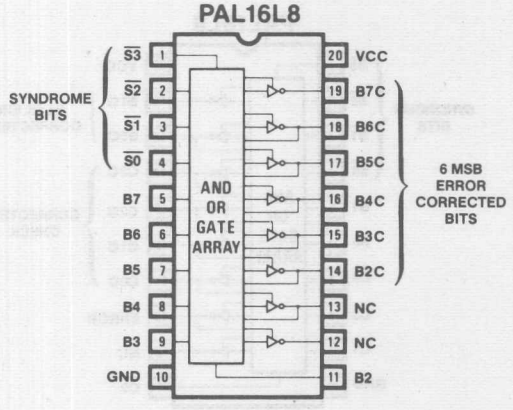
Description

This PAL device generates the syndrome bits for a 12 bit hamming code word as a function of the 8 data bits and the 4 check bits to point to any single bit in error.



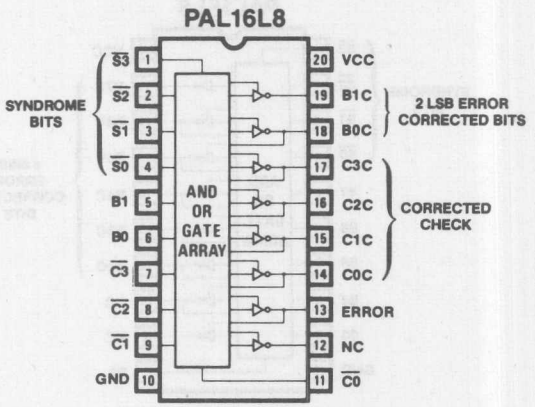
Description

This PAL device performs error correction of bits B2–B7 based on the 4 bit error syndrome S0–S3.



Description

This PAL device performs error correction of bits B0–B1 and checks bits C0–C3 based on the 4 bit error syndrome S0–S3.



```
PAL16X4                                          PAL DESIGN SPECIFICATION
CBG                                                   B. BRAFMAN 02/16/81
CHECK BIT GENERATOR
MMI FIELD APPLICATIONS ENGINEER YORBA LINDA, CALIFORNIA
CHKCLK B7 B6 NC NC NC NC B5 B4 GND
/OC B3 B2 /C0 /C1 /C2 /C3 B1 B0 VCC


C3 :=   B7*/B6                  ;B7 :+: B6
    +  /B7* B6                  ;   :+:
    +   GND                     ;DO NOT BLOW THIS PRODUCT LINE
    +   GND                     ;DO NOT BLOW THIS PRODUCT LINE
   :+:  B5*/B4                  ;B5 :+: B4
    +  /B5* B4

C2 :=   B7*/B3                  ;B7 :+: B3
    +  /B7* B3                  ;   :+:
    +   GND                     ;DO NOT BLOW THIS PRODUCT LINE
    +   GND                     ;DO NOT BLOW THIS PRODUCT LINE
   :+:  B2*/B1                  ;B2 :+: B1
    +  /B2* B1

C1 :=   B6* B5* B3              ;B6 :+: B5 :+: B3
    +  /B6*/B5* B3
    +  /B6* B5*/B3              ;   :+:
    +   B6*/B5*/B3
   :+:  B2*/B0                  ;B2 :+: B0
    +  /B2* B0

C0 :=   B6* B4* B3              ;B6 :+: B4 :+: B3
    +  /B6*/B4* B3
    +  /B6* B4*/B3              ;   :+:
    +   B6*/B4*/B3
   :+:  B1*/B0                  ;B1 :+: B0
    +  /B1* B0
```

```
PAL16X4                                          PAL DESIGN SPECIFICATION
SBG                                                  B. BRAFMAN 03/13/81
SYNDROME BIT GENERATOR
MMI FIELD APPLICATIONS ENGINEER YORBA LINDA, CALIFORNIA
SYNCLK D7 D6 B0 B1 B2 B3 D5 D4 GND
/OC D3 D2 A3 A2 A1 A0 D1 D0 VCC


;    IN THE ABOVE PIN LIST, THE FOLLOWING SUBSTITUTIONS HAVE BEEN
;    MADE TO ACCOMODATE THE SPECIFIC FORMAT (FIXED SYMBOLS) FOR THE
;    ARITHEMETIC PAL DEVICES IN PALASM SOFTWARE :

;         D7 MEANS B7        B0 MEANS /C3  (CHECK BIT 3)
;         D6 MEANS B6        B1 MEANS /C2  (CHECK BIT 2)
;         D5 MEANS B5        B2 MEANS /C1  (CHECK BIT 1)
;         D4 MEANS B4        B3 MEANS /C0  (CHECK BIT 0)
;         D3 MEANS B3        A0 MEANS /S3  (SYNDROME BIT 3)
;         D2 MEANS B2        A1 MEANS /S2  (SYNDROME BIT 2)
;         D1 MEANS B1        A2 MEANS /S1  (SYNDROME BIT 1)
;         D0 MEANS B0        A3 MEANS /S0  (SYNDROME BIT 0)

;  B0-B7 ARE THE BITS OF THE DATA WORD.
;  THE SUBSTITUTIONS APPLY BELOW WITH THE EXCEPTION OF COMMENTS.


/A0 :=    D7* D6* D5              ;B7 :+: B6 :+: B5
     +   /D7*/D6* D5
     +   /D7* D6*/D5              ;   :+:
     +    D7*/D6*/D5
    :+:   D4*(/B0)                ;B4 :+: C3
     +   /D4*( B0)

/A1 :=    D7* D3* D2              ;B7 :+: B3 :+: B2
     +   /D7*/D3* D2
     +   /D7* D3*/D2              ;   :+:
     +    D7*/D3*/D2
    :+:   D1*(/B1)                ;B1 :+: C2
     +   /D1*( B1)

/A2 :=    D6* D5* D3              ;B6 :+: B5 :+: B3
     +   /D6*/D5* D3
     +   /D6* D5*/D3              ;   :+:
     +    D6*/D5*/D3
    :+:   D2* D0*( B2)            ;B2 :+: B0 :+: C1
     +   /D2*/D0*( B2)
     +   /D2* D0*(/B2)
     +    D2*/D0*(/B2)

/A3 :=    D6* D4* D3              ;B6 :+: B4 :+: B3
     +   /D6*/D4* D3
     +   /D6* D4*/D3              ;   :+:
     +    D6*/D4*/D3
    :+:   D1* D0*( B3)            ;B1 :+: B0 :+: C0
     +   /D1*/D0*( B3)
     +   /D1* D0*(/B3)
     +    D1*/D0*(/B3)
```

```
TITLE     ERROR CORRECTION UNIT NO. 1
PATTERN   ECU1
REVISION  01
AUTHOR    B. BRAFMAN
COMPANY   MONOLITHIC MEMORIES, INC.
DATE      11/06/87

CHIP ECU1 PAL16L8

/S3 /S2 /S1 /S0 B7  B6  B5  B4  B3  GND
 B2  NC  NC B2C B3C B4C B5C B6C B7C VCC

EQUATIONS

/B7C =  S3* S2*/S1*/S0* B7          ;CORRECTION OF B7
     + /S3*/B7                      ;NO CORRECTION
     + /S2*/B7                      ;NO CORRECTION
     +  S1*/B7                      ;NO CORRECTION
     +  S0*/B7                      ;NO CORRECTION

/B6C =  S3*/S2* S1* S0* B6          ;CORRECTION OF B6
     + /S3*/B6                      ;NO CORRECTION
     +  S2*/B6                      ;NO CORRECTION
     + /S1*/B6                      ;NO CORRECTION
     + /S0*/B6                      ;NO CORRECTION

/B5C =  S3*/S2* S1*/S0* B5          ;CORRECTION OF B5
     + /S3*/B5                      ;NO CORRECTION
     +  S2*/B5                      ;NO CORRECTION
     + /S1*/B5                      ;NO CORRECTION
     +  S0*/B5                      ;NO CORRECTION

/B4C =  S3*/S2*/S1* S0* B4          ;CORRECTION OF B4
     + /S3*/B4                      ;NO CORRECTION
     +  S2*/B4                      ;NO CORRECTION
     +  S1*/B4                      ;NO CORRECTION
     + /S0*/B4                      ;NO CORRECTION

/B3C = /S3* S2* S1* S0* B3          ;CORRECTION OF B3
     +  S3*/B3                      ;NO CORRECTION
     + /S2*/B3                      ;NO CORRECTION
     + /S1*/B3                      ;NO CORRECTION
     + /S0*/B3                      ;NO CORRECTION

/B2C = /S3* S2* S1*/S0* B2          ;CORRECTION OF B2
     +  S3*/B2                      ;NO CORRECTION
     + /S2*/B2                      ;NO CORRECTION
     + /S1*/B2                      ;NO CORRECTION
     +  S0*/B2                      ;NO CORRECTION

; SIMULATION NOT INCLUDED
```

```
TITLE     ERROR CORRECTION UNIT NO. 2
PATTERN   ECU2
REVISION  01
AUTHOR    B. BRAFMAN
COMPANY   MONOLITHIC MEMORIES, INC.
DATE      11/06/87

CHIP ECU2 PAL16L8

/S3 /S2 /S1   /S0 B1  B0 /C3 /C2 /C1  GND
/C0 NC  ERROR C0C C1C C2C C3C B0C B1C VCC

EQUATIONS

/B1C = /S3* S2*/S1* S0* B1          ;CORRECTION OF B1
     +  S3*/B1                      ;NO CORRECTION
     + /S2*/B1                      ;NO CORRECTION
     +  S1*/B1                      ;NO CORRECTION

/B0C = /S3*/S2* S1* S0* B0          ;CORRECTION OF B0
     +  S3*/B0                      ;NO CORRECTION
     +  S2*/B0                      ;NO CORRECTION
     + /S1*/B0                      ;NO CORRECTION
     + /S0*/B0                      ;NO CORRECTION

/C3C =  S3*/S2*/S1*/S0* C3          ;CORRECTION OF C3
     + /S3*/C3                      ;NO CORRECTION
     +  S2*/C3                      ;NO CORRECTION
     +  S1*/C3                      ;NO CORRECTION
     +  S0*/C3                      ;NO CORRECTION

/C2C = /S3* S2*/S1*/S0* C2          ;CORRECTION OF C2
     +  S3*/C2                      ;NO CORRECTION
     + /S2*/C2                      ;NO CORRECTION
     +  S1*/C2                      ;NO CORRECTION
     +  S0*/C2                      ;NO CORRECTION

/C1C = /S3*/S2* S1*/S0* C1          ;CORRECTION OF C1
     +  S3*/C1                      ;NO CORRECTION
     +  S2*/C1                      ;NO CORRECTION
     + /S1*/C1                      ;NO CORRECTION
     +  S0*/C1                      ;NO CORRECTION

/C0C = /S3*/S2*/S1* S0* C0          ;CORRECTION OF C0
     +  S3*/C0                      ;NO CORRECTION
     +  S2*/C0                      ;NO CORRECTION
     +  S1*/C0                      ;NO CORRECTION
     + /S0*/C0                      ;NO CORRECTION

/ERROR = /S3*/S2*/S1*/S0            ;NO ERROR!

; SIMULATION NOT INCLUDED
```

Fuse Programmable Controller Simplifies Cache Design

Introduction

The growing sophistication of computer systems has dictated a corresponding increase in memory storage capacity. Unfortunately, the speed of these large memories has not kept pace with that of their associated processors. Fast memories are still expensive; in order to reduce this cost, a cache memory system can be used to provide information to the processor without sacrificing speed. Located between the central processing unit and the main memory a cache memory stores only the blocks of memory currently in use by a program. This provides fast access to program and data resulting in higher system performance. Once a new memory location is desired, blocks of cache and main memory are swapped to update the cache with the memory blocks currently in use. Thus a cache memory can dramatically increase the performance of the entire computer system while maintaining minimal costs. A cache system with a 35 ns access time and a 50 ns cycle time is described below.

The use of a fuse-programmable controller (FPC), Am29PL141, simplifies the design of such a cache memory system. The simple internal architecture and instruction set of the FPC make this controller easy to implement in systems, and an on-chip microprogrammable fuse array provides a high level of integration. The cache controller algorithm is programmed and stored in this array.

Because the FPC operates at 20 MHz, as opposed to the 10 MHz clock frequency of other currently available controllers, it is especially well suited for high-performance applications.

Cache Memory Systems

Almost any system that requires a high-speed memory interface can benefit from the use of a cache-memory system. The system bottleneck is typically the memory access time. Large dynamic RAMs that are currently available have access times of 100–180 ns with cycle times of about 180–400 ns. The smaller, faster memory devices that can be used for a cache have access times of 25–30 ns and cycle times of around 40–50 ns. In order to achieve these lower access times and still have the memory capacity of the larger DRAM memories, a cache memory hierarchy is used.

Any application can make effective use of caches provided that the concept of "locality of reference" is operative. This concept states that, at any particular time, the addressing pattern of a program tends to be localized within a block or area of the total available address space. In other words, if data at a given address is accessed, it is likely that the next memory access will also be in that same block of addresses. In general, for systems where this concept does not apply, the use of a cache system would be difficult.

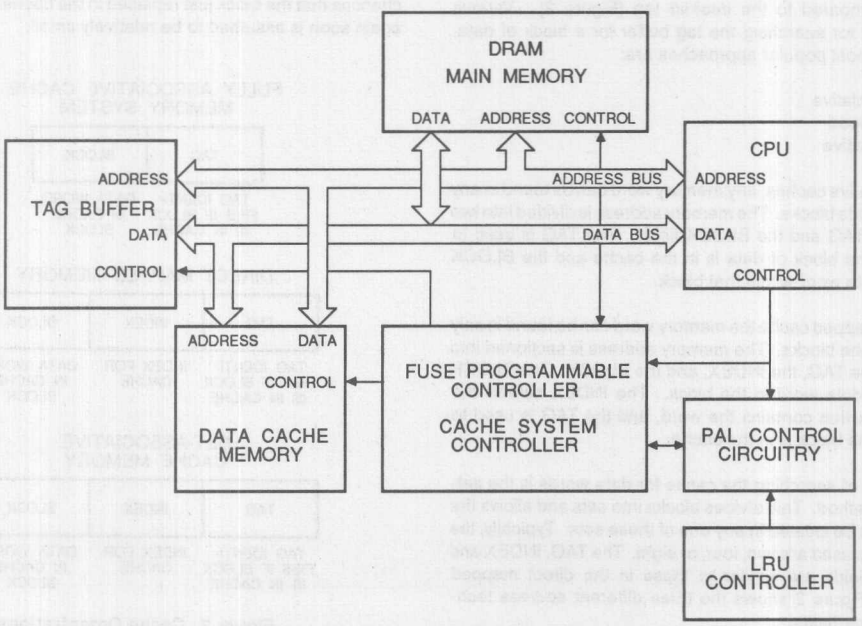


Figure 1. Block Diagram of Cache Memory System

Sections of main memory data that can be identified as frequently used can be stored in cache memories to allow decreased access times, thereby increasing the overall performance of the system. As different sections of a program become frequently used, they can be transferred to the cache memory, displacing other sections that are no longer needed. Some of the application areas where caches are used to increase the system performance are for mini/micro-computers, signal processing and image processing systems.

The main components of the cache memory system are the cache data memory, the cache tag buffer, the replacement logic, and the cache controller (see Figure 1).

Cache Data Memory

The cache data memory is the small, fast memory that contains the most often used data, copied from the slower main memory. The cache data memory size directly affects the performance of the entire system. The larger the cache data memory, the more data that can be accessed at the faster speeds. However, if the size of the memory device gets very large, the performance of the memory decreases. The larger the memory devices are, the longer are their access times, and, consequently, their cycle times. The optimum data cache size will vary depending on the application/algorithm being executed.

Tag Buffer

An identifier (or tag) accompanies every block (a contiguous set of data words) that can be transferred to or from main memory. This tag is stored in the tag buffer. This tag identifies where the data is located in main memory. To determine whether a block of data is already in the cache memory, the tags stored in the tag buffer are compared to the desired tag (Figure 2). Various methods exist for searching the tag buffer for a block of data. Three of the most popular approaches are:

1. Fully associative
2. Direct mapped
3. Set-associative

In fully associative caches, any memory word can be found in any of the cache-data blocks. The memory address is divided into two sections: the TAG and the BLOCK fields. The TAG is used to determine if the block of data is in the cache and the BLOCK indicates a data word within that block.

For a direct-mapped cache the memory word can be found in only one of the cache blocks. The memory address is sectioned into three fields: the TAG, the INDEX, and the BLOCK. The BLOCK indicates the data word in the block. The INDEX selects the program block that contains the word, and the TAG is used to determine if the block is in the cache.

The third way of searching the cache for data words is the set-associative method. This divides blocks into sets and allows the cache block to be located in any one of these sets. Typically, the number of sets used are two, four, or eight. The TAG, INDEX and the BLOCK fields are similar to those in the direct mapped techniques. Figure 2 shows the three different address techniques and their fields.

When a tag search is performed in the tag buffer, either a hit or miss condition results. A hit indicates that the desired word is in the cache; a miss indicates that the desired word must be accessed from the main memory or a transfer from main memory to cache is needed. The performance of the cache system is dependent on the ratio of the number of hits to the total number of memory references made, which is called the hit ratio. The hit ratio depends on the size of the cache memory selected and the application of the system using the cache memory. A high hit ratio implies that most of the data accessed by the CPU was in the fast data-cache memory when requested. The performance of the entire system gets degraded if a poor hit ratio occurs. This is why "locality of reference" is important for performance.

Replacement Algorithms

The replacement algorithm is used to decide the cache location to replace with new data when a miss occurs. Three of the most common data replacement algorithms currently in use are:

1. First-in First-out (FIFO)
2. Least Recently Used (LRU)
3. Random Replacement (RR).

The choice between these three techniques is dependent upon the nature of the data being transferred to and from the cache. The FIFO algorithm simply selects the oldest block of data written into the cache to be replaced. After the cache buffer is full, data blocks are replaced sequentially. In this manner, even a data block which is used very often, will be replaced.

The LRU technique selects the data block to be replaced by determining which block in the cache was least used as compared to the other blocks at that time. With this strategy, the chances that the block just replaced in the cache will be required again soon is assumed to be relatively small.

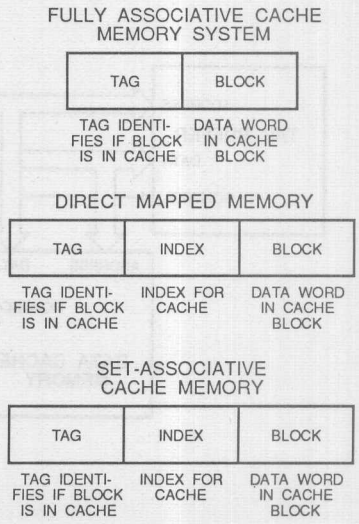


Figure 2. Cache Organizations

The third method, the random replacement approach, randomly chooses the block of data to be replaced. The easiest to implement, this technique is optimum for some applications.

Another important design consideration for cache memory systems is the method used for keeping the cache data and main memory updated. When writing to the cache memory, the data change must be echoed to the main memory before that data word is replaced by new data in the cache block. One way this is handled is the "store-through" method. On a write command, the data is written to the cache as well as the main memory simultaneously. In this way, the main memory data is kept current. This also saves the task of writing back old cache data into main memory when replacing a new data block into cache.

Cache Controller

All of the signals that are required to keep track of the various functional blocks involved in the cache memory system are controlled by the cache controller. This controller monitors the state of the tag buffer search, the CPU, main memory and cache data memory condition, as well as handling all memory transfers for hits or misses and read and write operations. The high-performance and integrated functions available on a single chip make the design of a cache controller using the Am29PL141 simple. The FPC provides 16 control signals for off-chip control and seven input test pins for monitoring external conditions. The flexible instruction set permits easy microprogramming of the desired function.

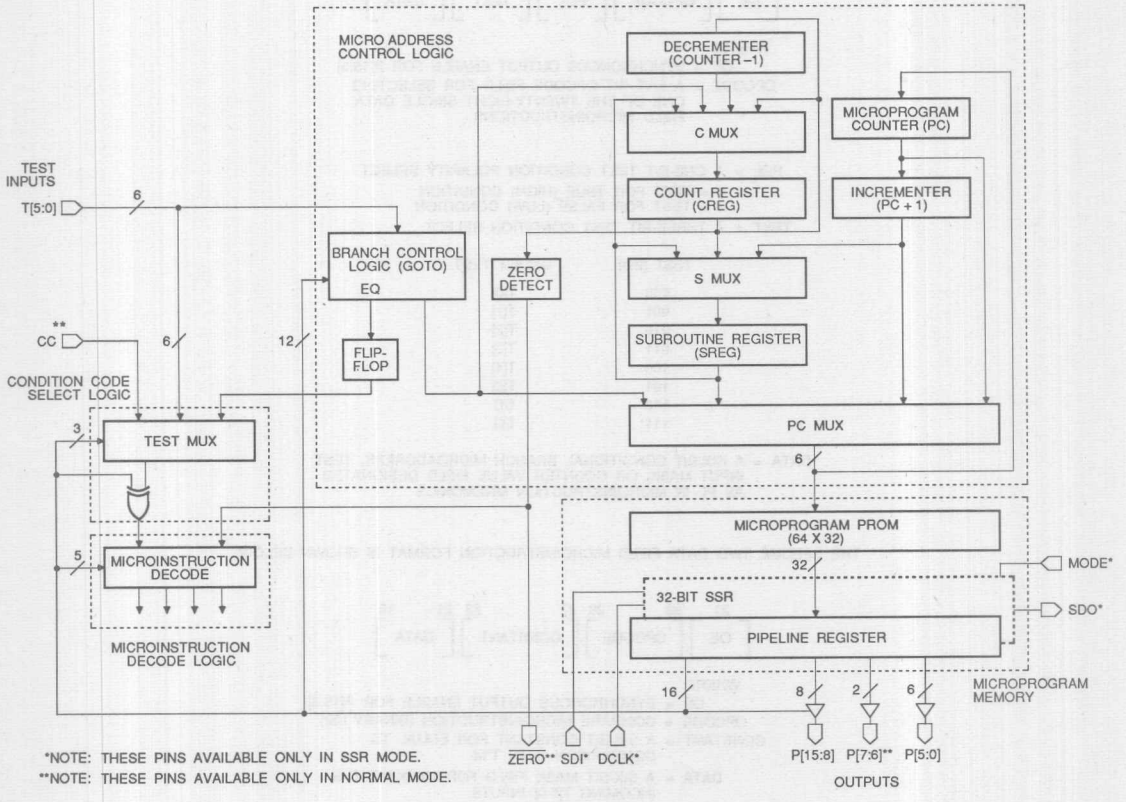
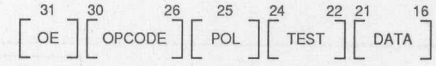


Figure 3. FPC Detailed Functional Diagram

The Am29PL141 Fuse-Programmable Controller

The Am29PL141 is a high-performance, single-chip, fuse-programmable controller (FPC). This chip is designed to allow implementation of complex state machines and controllers by programming the appropriate sequence of microinstructions in the on-chip microprogram memory. With its intelligent microprogram address sequencer, high-speed 64 x 32 bit microprogram memory, pipeline register, and an on-chip diagnostics register, this chip offers the benefits of low chip count, fast operation, easy development, and testability. This device is available with a 20 MHz clock rate (50 ns cycle time) in a 28 pin dual-in-line package. A microprogram address sequencer is the heart of the FPC. The Am29PL141 has 29 high-level microinstructions which include jumps, loops, subroutine calls, and multiway branching. These microinstructions can be conditionally executed based on the test inputs. As a single chip solution, the FPC is designed to ease the implementation of distributed microprogrammed systems. It can be used to off-load the central controller by serving as an intelligent distributed controller for various functional units. Its on-chip diagnostics register is used to control and observe the parallel pipeline register during diagnostics mode. This capability provides both controllability and observability for the functional blocks connected to the FPC control outputs. A block diagram of the FPC is shown in Figure 3 and the micro-instruction format is shown in Figure 4.



WHERE:

OE = SYNCHRONOUS OUTPUT ENABLE FOR P[15:8]

OPCODE = A FIVE-BIT OPCODE FIELD FOR SELECTING ONE OF THE TWENTY-EIGHT SINGLE DATA FIELD MICROINSTRUCTIONS

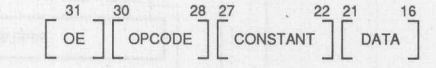
POL = A ONE-BIT TEST CONDITION POLARITY SELECT
0 = TEST FOR TRUE (HIGH) CONDITION
1 = TEST FOR FALSE (LOW) CONDITION

TEST = A THREE-BIT TEST CONDITION SELECT

TEST [2:0]	UNDER TEST
000	T[0]
001	T[1]
010	T[2]
011	T[3]
100	T[4]
101	T[5]
110	CC
111	EQ

DATA = A SIX-BIT CONDITIONAL BRANCH MICROADDRESS, TEST INPUT MASK, OR COUNTER VALUE FIELD DESIGNATED AS PL IN MICROINSTRUCTION MNEMONICS

THE SPECIAL TWO DATA FIELD MICROINSTRUCTION FORMAT IS SHOWN BELOW:



WHERE:

OE = SYNCHRONOUS OUTPUT ENABLE FOR P[15:8]

OPCODE = COMPARE MICROINSTRUCTION (BINARY 100)

CONSTANT = A SIX-BIT CONSTANT FOR EQUAL TO COMPARISON WITH T*M

DATA = A SIX-BIT MASK FIELD FOR MASKING THE INCOMING T[5:0] INPUTS

Figure 4. Microinstruction Format

442 04

System Overview

The cache scheme selected for implementation is a set-associative cache with four sets (see Figure 5). The 18-bit physical memory address word is divided into 8 bits of tag, 8 bits of index, and 2 bits of block address. Four 256-byte deep, index-addressed tag buffers are used in this scheme. These tag buffers contain four possible sets of tags for the same index location. Corresponding to each tag buffer there are four 16-bit wide cache memories, which contain four (block) words for each index location. The physical block address is also used to address these cache memories.

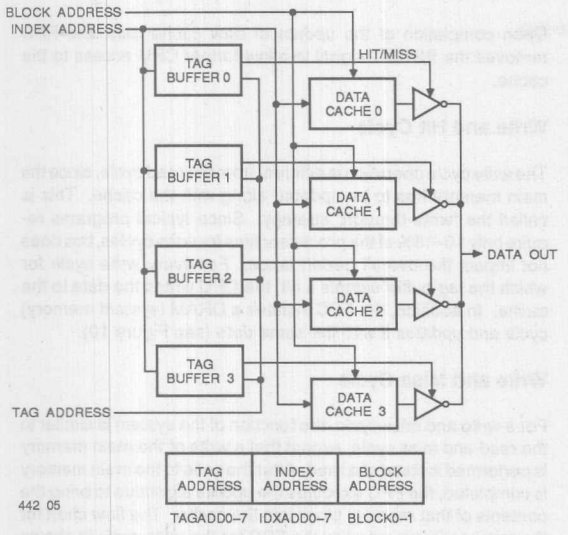


Figure 5. A Set-Associative Cache Scheme

Figure 6 shows the actual implementation where the main system controller is the Am29PL141 FPC. There is additional glue logic in the form of one PAL18P8 and one PAL16R6. The PAL18P8 is used in the cache access path and is very fast. It generates the cache output enable, cache selects, and write signals for the cache. The PAL16R6 is used to generate the Ready signal to the system CPU. It is also used to inform the FPC of the status of read/write cycles. All the PAL devices are controlled by the FPC for generation of the system control signals.

System Operation

There are four main system cycles possible: read and hit, read and miss, write and hit and write and miss. The response of the controller is different for each system operation cycle. For read & hit cycle the FPC performs no action. Only on detection of write cycle, or read & miss cycle, the PAL16R6 asserts the CYCL signal initiating FPC action.

Read and Hit Cycle

When the system CPU requests a read cycle, it sends out an 18-bit address. The most significant eight bits (Figure 6) TAGADD0–7 are the eight bit tag which is compared to the tags stored in the four tag buffers to detect a hit or a miss. The next eight bits IDXADD0–7 for index, address the correct tag buffer

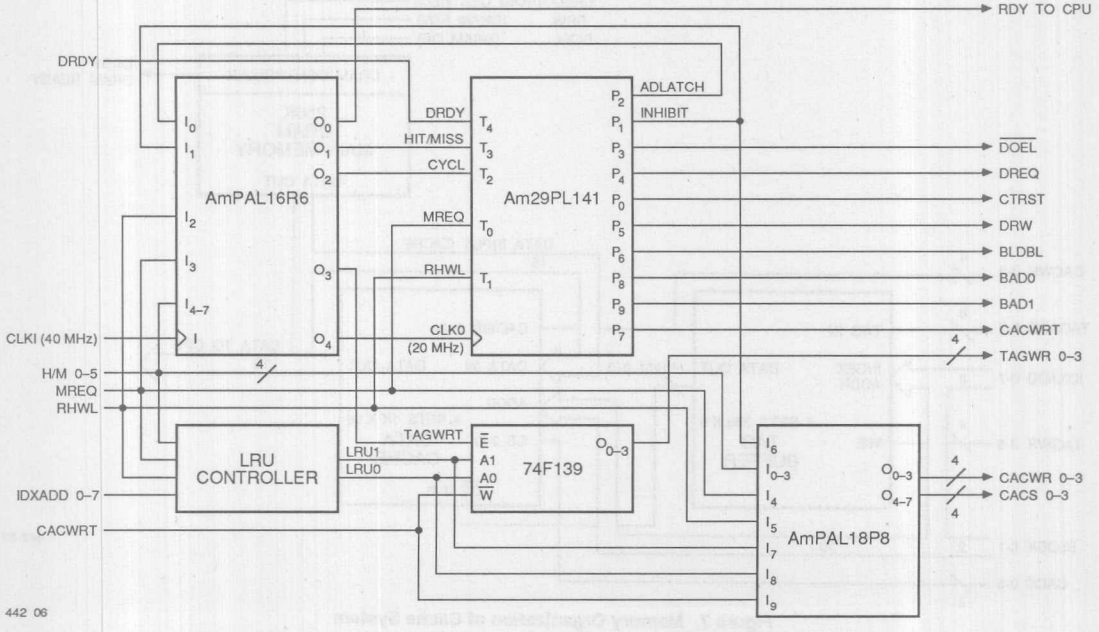


Figure 6. Control Section for Cache System uses one FPC and two PAL Devices.

locations. These eight bits are also used to access the four cache sets simultaneously. The least significant BLOCK0–1 block address are directly used to address the four cache sets. A cache CACS (chip select) is generated by the PAL18P8 to access the four cache sets simultaneously. Once the tag buffer match occurs, the selected tag buffers asserts the hit signal. The hit signal is used to enable the outputs of the corresponding cache set, in order to provide the correct cache data to the CPU. The flow chart is shown in Figure 9.

Read and Miss Cycle

If the proper tag does not appear in any of the four tag buffers, a miss signal is generated by the tag buffers on H/M0–3 signal lines (Figure 7 and 8). On detection of the four miss signals the PAL16R6 removes the "RDY" signal to CPU (for CPU wait cycles) and informs the FPC of the MISS by asserting the CYCL & HIT/MISS signals. When the FPC detects this read and miss, it latches the current physical address by asserting the ADLATCH signal, and initiates a DRAM (System Memory) cycle by asserting the DREQ signal, to provide the data to the CPU from the system memory. When this data is available signalled by the DRAM data ready signal DRDY, the FPC signals the assertion of the RDY signal, allowing the CPU to read the requested data. Simultaneously the FPC asserts the INHIBIT signal which disallows further CPU access to the cache.

The FPC then performs an update algorithm which brings the data from the system memory to the cache. It asserts the CTRST signal which disables the CPU data lines to the cache system. It also disables CPU block addresses by asserting BLDBL signal and enables its own output address lines BAD0–1 in order to address different memory words within a block itself. It then performs read cycles on the system memory by using DREQ (DRAM memory request), DRW (read or write) and DRDY (DRAM data ready) signals and write cycles on all the blocks of the corresponding cache set by asserting CACWRT signal. The replacement set is determined by the LRU control block which selects one of the four sets.

Upon completion of the update of new cache data, the FPC removes the INHIBIT signal to allow further CPU access to the cache.

Write and Hit Cycle

The write cycle operation is different from the read cycle, since the main memory has to be updated along with the cache. This is called the "write-through" strategy. Since typical programs require only 10–15% of the processor time for write cycles, this does not impact the overall performance. For every write cycle for which the tag buffer signals a hit, the CPU writes the data to the cache. In addition, the FPC initiates a DRAM (system memory) cycle and updates it with the same data (see Figure 10).

Write and Miss Cycle

For a write and miss cycle, the function of the system is similar to the read and miss cycle, except that a write of the main memory is performed instead of a read. After the write to the main memory is completed, the FPC executes the update algorithm to bring the contents of that memory block into the cache. The flow chart for the microcode executed by the FPC for the write cycle is shown in Figure 10.

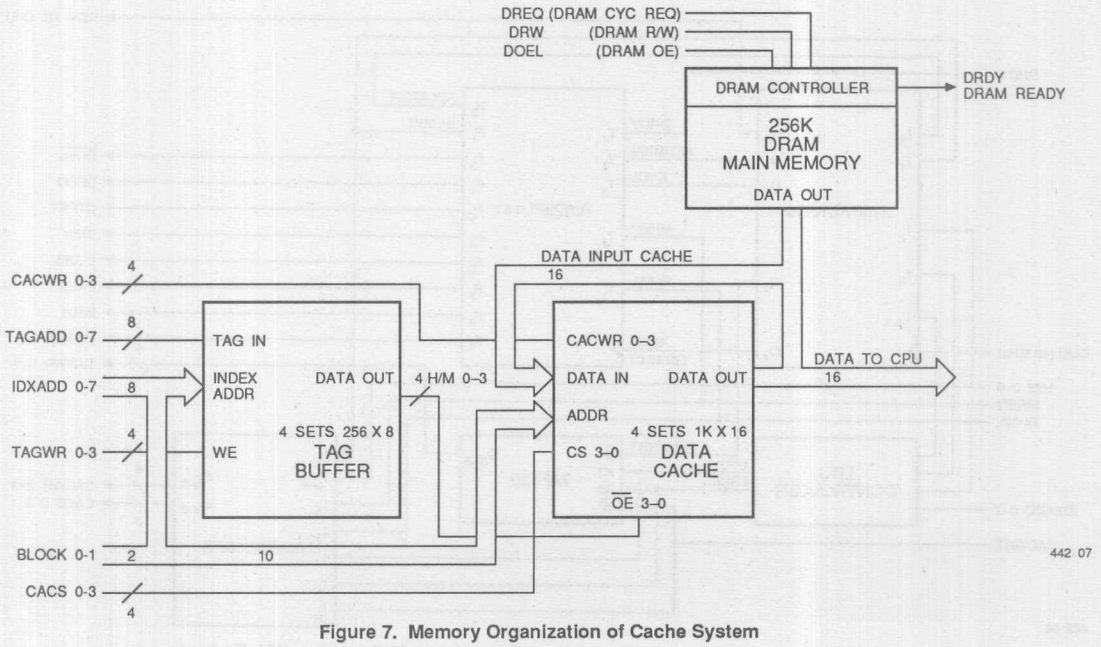


Figure 7. Memory Organization of Cache System

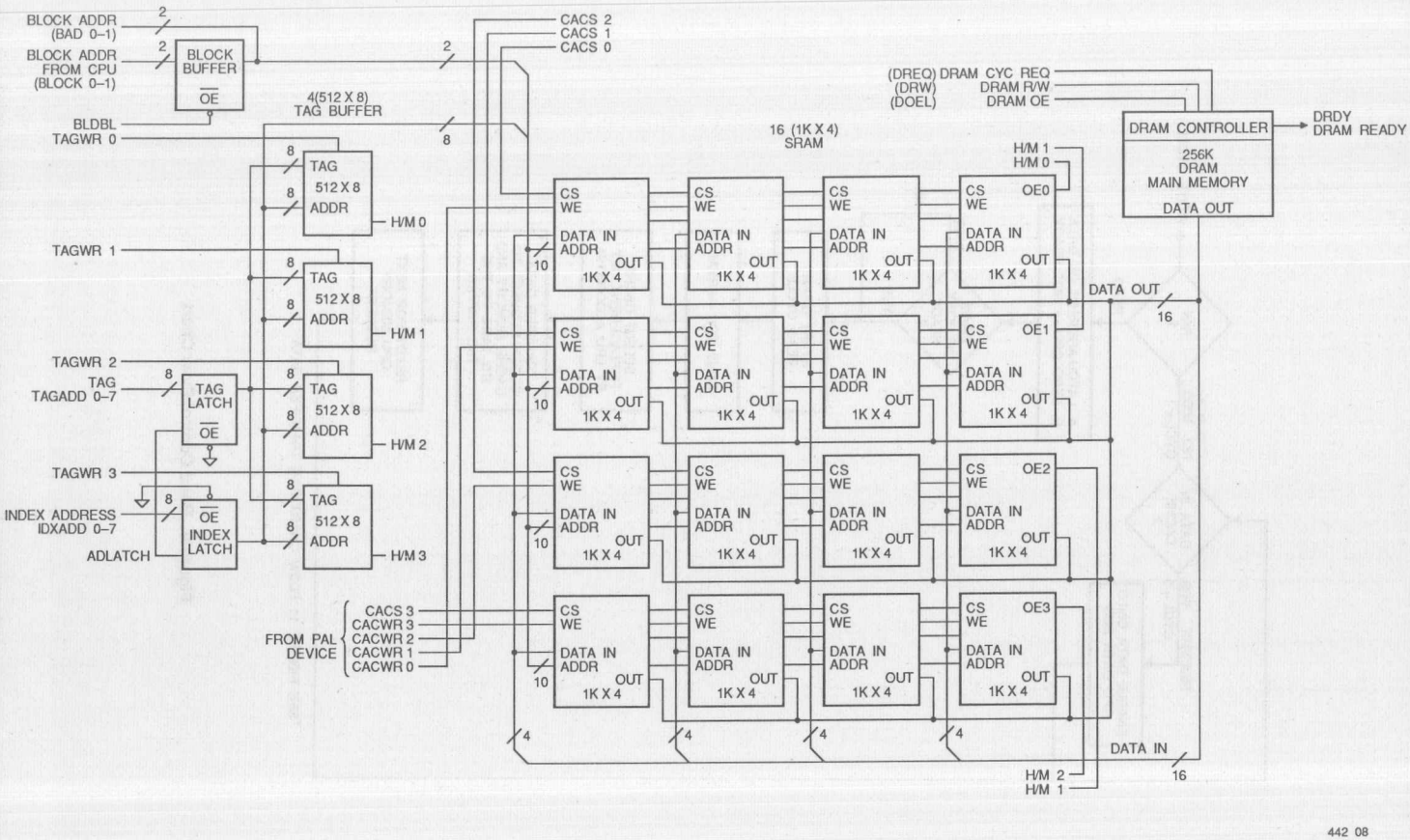
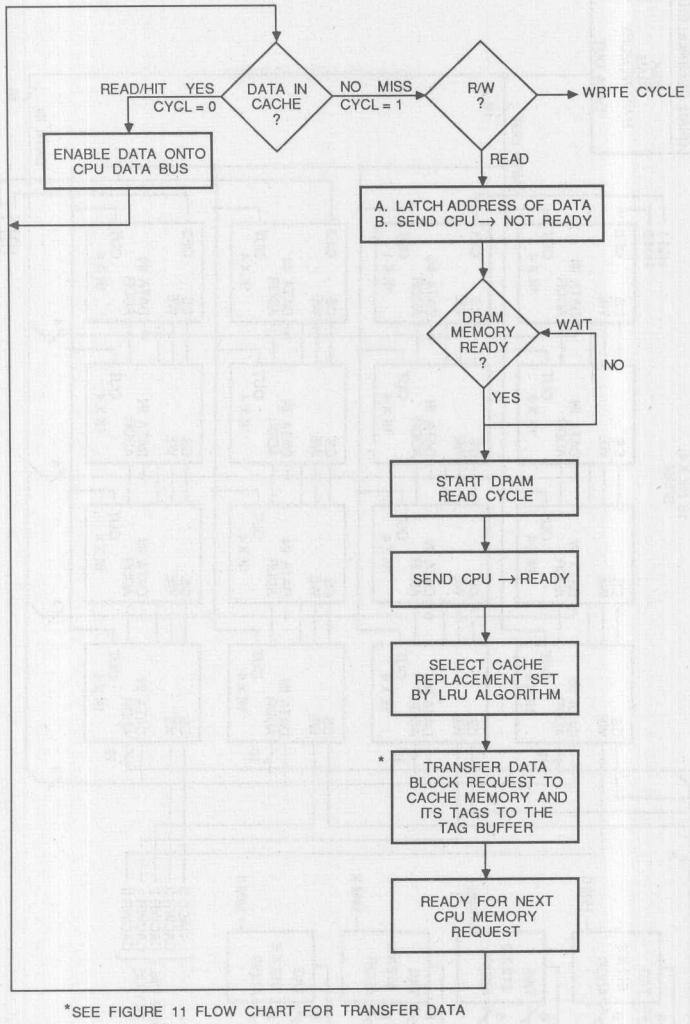
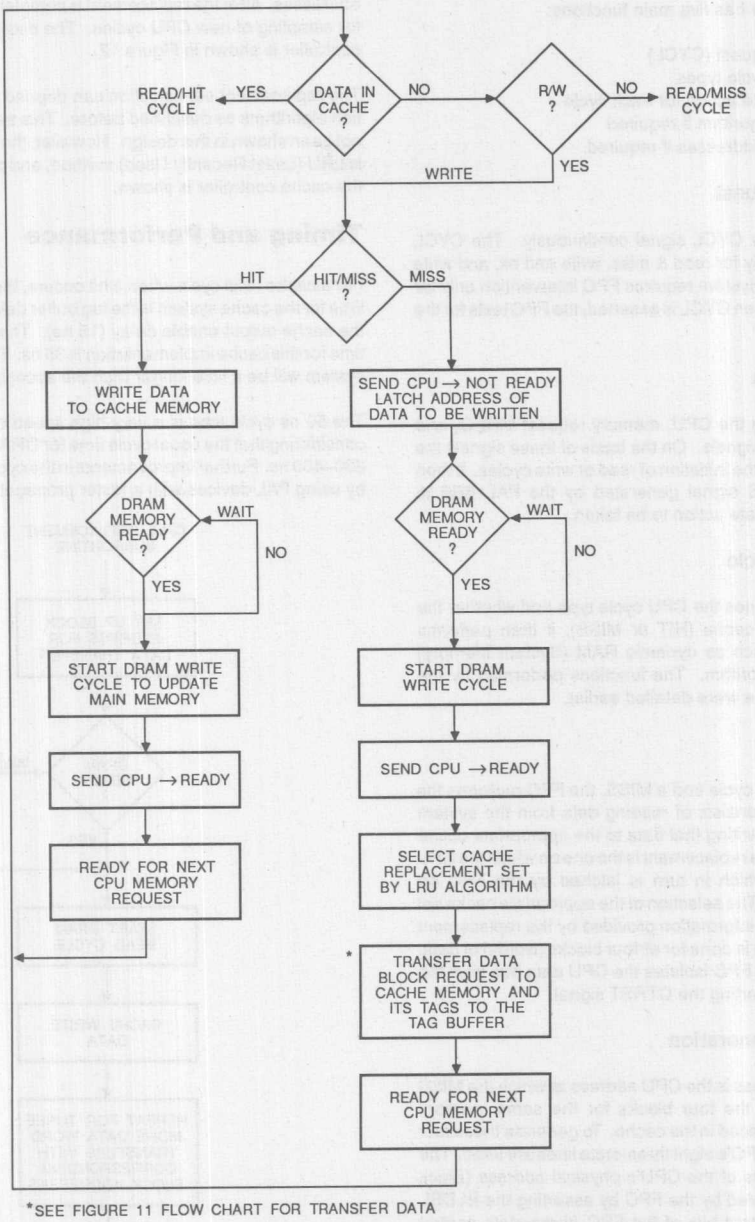


Figure 8. Detailed Diagram of Tag Buffer and Data Cache



442 09

Figure 9. Read Control Flow Chart



442 10

Figure 10. Write Control Flow Chart

FPC Operation

The System Controller has five main functions:

1. Test for a cycle request (CYCL)
2. Test for different cycle types
3. Execute appropriate action for each cycle
4. Execute update algorithm if required
5. Generate update addresses if required.

Test for Cycle Request

The FPC samples the CYCL signal continuously. The CYCL signal is generated only for read & miss, write and hit, and write and miss cycles. The system requires FPC intervention only for these three cycles. When CYCL is asserted, the FPC tests for the three cycle types.

Test for Cycle Type

This involves sampling the CPU, memory request MREQ, and read or write (RHWL) signals. On the basis of these signals the FPC decides between the initiation of read or write cycles. It then samples the HIT/MISS signal generated by the PAL16R6 to decide on the appropriate action to be taken.

Action for Each Cycle

Once the FPC determines the CPU cycle type and whether the location exists in the cache (HIT or MISS), it then performs appropriate actions such as dynamic RAM (System Memory) cycle, and Update algorithm. The functions performed by the FPC for each cycle type were detailed earlier.

Update Algorithm

On detection of a CPU cycle and a MISS, the FPC performs the update algorithm. It consists of reading data from the system memory (DRAM) and writing that data to the appropriate cache set. The address for this replacement is the one on which the CPU cycle MISS occurs, which in turn is latched by the FPC on detection of the MISS. The selection of the appropriate cache set is dependent upon the information provided by the replacement logic block. The update is done for all four blocks (words) of data. During this update, the FPC isolates the CPU data bus from the cache data bus by asserting the CTRST signal.

Update Address Generation

The replacement address is the CPU address at which the MISS occurs. However, all the four blocks for the same memory address should be replaced in the cache. To generate these four addresses, two of the FPC's eight three-state lines are used. The two least significant bits of the CPU's physical address (Block Address bits) are disabled by the FPC by asserting the BLDBL signal, and the two output bits of the FPC (three-state control lines) are enabled onto the address bus. This is done in a manner to allow the system CPU to continue to work with the address and data bus, isolated from the cache system, during the execution of the replacement algorithm.

The replacement algorithm then executes read cycles on the memory and write cycles on the cache for all the four block addresses. After the replacement is complete, the control returns for sampling of new CPU cycles. The design file for the Cache controller is shown in Figure 12.

The replacement set selection can depend upon various selection algorithms as described before. This set selection logic has not been shown in this design. However, the algorithm used here is LRU (Least Recently Used) method, and only its interface with the cache controller is shown.

Timing and Performance

For a cache read cycle when a hit occurs, the worst case access time for the cache system is the tag buffer delay (20 ns) along with the cache output enable delay (15 ns). The worst case access time for this cache implementation is 35 ns. The cycle time for the system will be a little longer than the access time.

The 50 ns cycle time is a very high speed cache performance, considering that the usual cycle time for DRAMs is in the range of 280–400 ns. Further improvements in the cycle time can be made by using PAL devices with a faster propagation delay of 10 ns.

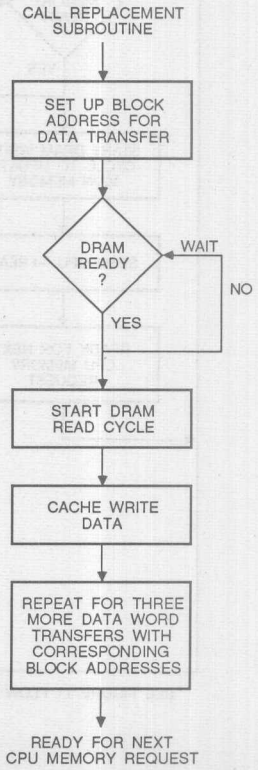


Figure 11. Flow Chart for Cache Update Scheme

```
device (am29pl141)
default = 0 ;

define
   mreq   = t0
   rhwl   = t1
   hitm   = t2
   drdy   = t3
   nocdn  = t4                              "No condition"
   intword = 07d#h;                         "Initialize output word"

begin
init   : od intword, continue ;             "Initialize Outputs"

start  : od intword, if (mreq) then goto pl(start);
                                            "Check for memory cycle"
         od intword, if (not rhwl) then goto pl(write);
                                            "Check for read or write"

read   : od intword, if (hitm) then goto pl(start);
                                            "Read & hit goto start"

latch  : od 079#h,   if (not drdy) then goto pl(latch);
                                            "Read & miss latch address"
         od 069#h,   continue;              "DRAM req for CPU data"
         od 063#h,   continue;              "DRAM output enb & CPU cycle"
         od 063#h,   if (nocdn) then call pl(rplce);
                                            "Goto replacement subroutine"
         od intword, if (nocdn) then goto pl(start);
                                            "Read cycle complete"

write  : od intword, if (not hitm) then goto pl(wrtms);
                                            "Write hit or miss"

wrtht  : od 0fd#h,   if (not drdy) then goto pl(wrtht);
                                            "Hit write to cache"
         od 04d#h,   continue;              "Write to DRAM"
         od 04f#h,   continue;
         od 04f#h,   if (nocdn) then goto pl(start);
                                            "Write hit complete"

wrtms  : od intword, if (not drdy) then goto pl(wrtms);
                                            "Miss check DRAM for write"
         od 049#h,   continue;
         od 04b#h,   continue;              "Latch address & DRAM write"
         od 04b#h,   if (nocdn) then call pl(rplce);
                                            "goto replacement algorithm"
         od intword, if (nocdn) then goto pl(start);
                                            "Write miss complete"

rplce  : od 07b#h,   if (not drdy) then goto pl(rplce);
                                            "This is replace routine"
         oe 022#h,   continue;              "Isolate CPU  databus"
         oe 022#h,   continue;              "DRAM data enb for cache write"
         oe 0a2#h,   continue;              "Write to cache at address 00"
wait1  : od 13a#h,   if (not drdy) then goto pl(wait1);
                                            "Change Block address"
         oe 122#h,   continue;              "Repeat as before address 01"
         oe 122#h,   continue;
         oe 1a2#h,   continue;              "Write to cache at address 01"
wait2  : od 23a#h,   if (not drdy) then goto pl(wait2);
                                            "Change Block address"
         oe 222#h,   continue;              "Repeat as before address 02"
         oe 222#h,   continue;
         oe 2a2#h,   continue;              "Write to cache at address 02"
wait3  : od 33a#h,   if (not drdy) then goto pl(wait3);
                                            "Change Block address"
         oe 322#h,   continue;              "Repeat as before address 03"
         oe 322#h,   continue;
         oe 3a2#h,   continue;              "Write to cache at address 03"
         od 03a#h,   if (nocdn) then ret;"Return from replace routine"

end.
```

Figure 12. Cache Controller Design File

Addressing for a 32-Location-Deep RAM-Based LIFO

Two components used in the temporary storage of data are LIFO (Last In, First Out), and FIFO (First In, First Out) buffers. Usually, a FIFO would be used to store data between asynchronously operating devices. For example, data sent from one microprocessor system to a second might reside temporarily in a FIFO until the second microprocessor is ready to read the data. Sometimes the FIFO is known as an asynchronous temporary buffer store.

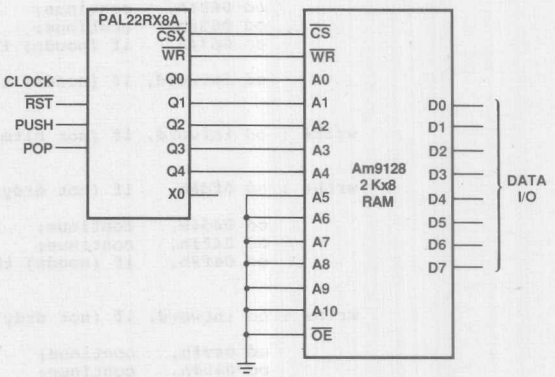
The LIFO also offers temporary storage and will store data in the same way as a FIFO but the sequence of reading the data will be reversed. LIFO applications in digital processing systems would usually be to store return addresses and essential registers when the processor is executing a branch to subroutine or interrupt service routine. The processor will retrieve this information when it executes return instructions. The LIFO is an essential component in the processor's ability to perform context switching.

The LIFO is sometimes called a stack, and the process of writing to the stack is called a PUSH operation and reading data is a POP operation. In the central processor dedicated instructions for PUSH and POP operations are decoded and executed to save and restore return addresses, contents of accumulators, status registers, and index registers. Other LIFO applications could be found in data buffering where the order of data reception should be reversed.

The concept of the stack is used in performing Reverse Polish Notation (RPN) mathematics, and it is claimed by some to be superior in performance to the conventional algebraic processing technique. With RPN, instructions and operands are pushed onto the stack, and evaluated as they are popped off; the result is usually left in the general purpose register.

When a memory is used as a stack, the evaluation process may be repeated because the initial evaluation did not destroy the RAM contents. This non-destructive reading of data is a feature of RAM-based FIFO and LIFO architectures, and is not the case for register-based architectures. FIFOs and LIFOs made from register arrays rely on data shuffling through the array. Data in the registers are overwritten during each cycle. The RAM control for a LIFO essentially consists of an Up/Down counter to address locations in the memory, and a control sequencer to drive the memory's chip select ($\overline{CS}$) and write ($\overline{WR}$) inputs.

The new PAL22RX8A has been programmed to perform a control function for a small 32-location-deep stack. A two-chip solution is achieved by using an Am 9128 static RAM and a PAL22RX8A (see Figure 1), for bytewide applications. Additional RAM devices may be added in parallel for 16- and 32-bit wide applications. A deeper LIFO may be configured by cascading a second PAL22RX8A to create a higher-order Up/Down counter.



RAM OPERATION

$\overline{WR}$	$\overline{CS}$	DATA
X	H	THREE-STATE
H	L	DATA OUT
L	L	DATA IN

Figure 1. 32-Location-Deep LIFO Controller Uses PAL22RX8A

Figure 2 shows the block diagram of the PAL22RX8A device. Fourteen dedicated inputs drive the single fuse array through true/complement buffers. One of those inputs includes the clock, so logic operations may be performed on the clock or its complement.

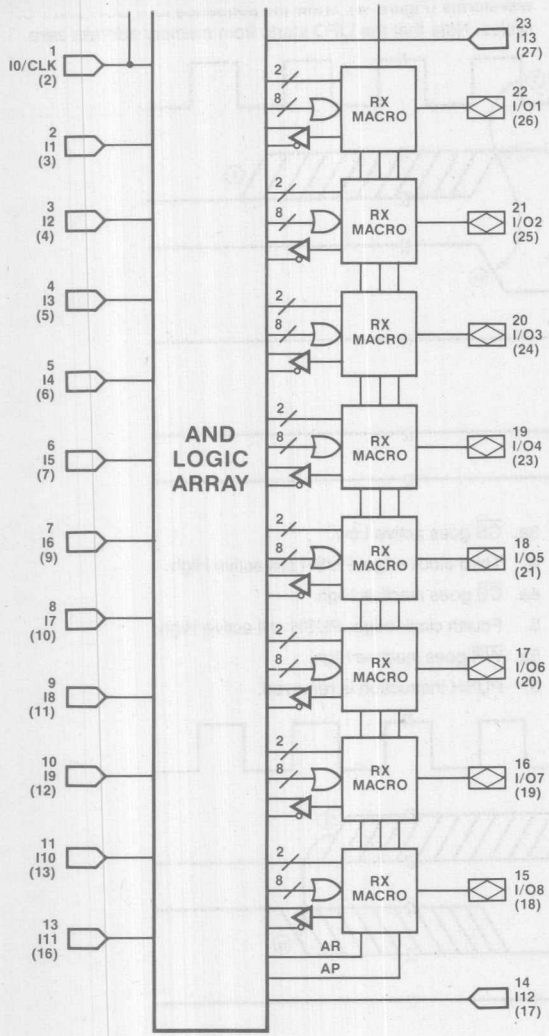


Figure 2. Block Diagram

Note:
PLCC pin numbers are indicated in parentheses.
PLCC pins 1, 8, 15 and 22 are not connected.

There exist eight macrocells in the device. The schematic view of the macrocell architecture is shown in Figure 3. The D-type register is the storage element in the device and is clocked from a single clock input. The D input is fed from an exclusive-OR (XOR) gate, the two inputs of which are driven from one product term and a sum of eight product terms. This XOR gate can be used to control the polarity at the output pin; the single product term input can be used to create polarity inversion.

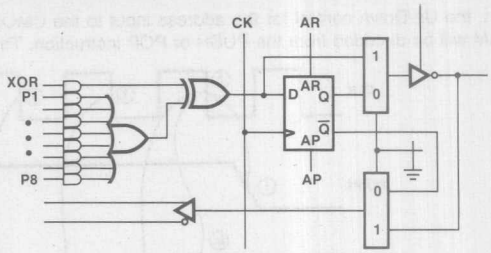


Figure 3. PAL22RX8A Macrocell

With regard to registered output configurations, the XOR gate may be used to configure alternative registered functions. It can be shown that a J-K flip-flop may be derived by using feedback, the sum-of-product inputs, and the XOR input, as shown in the equations below. Three out of the five equations use the XOR gate to create the J-K function.

1. $Q := \overline{Q}*J + Q*\overline{K}$
2. $\overline{Q} := \overline{Q}*\overline{J} + Q*K$
3. $Q := Q :+: (\overline{Q}*J + Q*K)$
4. $\overline{Q} := \overline{Q} :+: (\overline{Q}*J + Q*K)$
5. $Q := \overline{Q} :+: (\overline{Q}*\overline{J} + Q*\overline{K})$

Also, S-R and other subset functions may be configured in the macrocell.

There exist two multiplexers in the macrocell which select either a registered or combinatorial output path. If the control fuse is intact, the output is registered, and the feedback into the fuse array is through a buried feedback path from the $\overline{Q}$ node. A combinatorial route may be selected by programming the fuse; the register is flushed and feedback to the array is from the output pin. Programmable I/O may be achieved in this configuration by controlling the three-state condition of the inverting output buffer. A single product term can enable the output buffer with a logic High or disable it with a logic Low. When in a three-state condition the output pin may be used as an input.

When using the flip-flop in the macrocell it is possible to use the asynchronous Reset or Preset feature. A single input pin may be dedicated to a Reset or Preset function, and driven active to initialize the register in the device.

Toggle Function as a J-K Subset

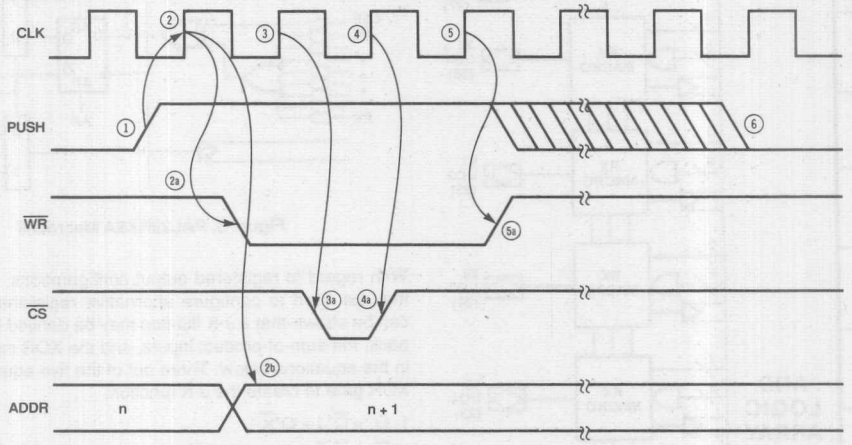
A frequently used subset of the J-K function used in binary count applications is the Toggle (T) function. If J and K are tied together and driven with a logic High then the flip-flop output will Toggle when clocked; if driven Low the output will Hold. For a given flip-flop (N) in the count sequence the toggle equation will be:

6. QN := QN :+: QN-1*QN-2*...Q1*Q0.

This will cause the flip-flop to Hold until the product term of all the lesser significant registers goes to a logic High; then, the flip-flop will toggle polarity. This equation is suitable for an Up count. A Down count is given by:

7. QN := QN :+: $\overline{QN\text{-}1}*\overline{QN\text{-}2}*..\overline{Q1}*\overline{Q0}$.

Both equations can be combined with an Up/Down control to effect an Up/Down counter. With regard to the LIFO application, the Up/Down control for the address input to the CMOS RAM will be decoded from the PUSH or POP instruction. The sequence will be "Increment then Write" for a PUSH operation and "Read then Decrement" for a POP operation. The timing waveforms (Figure 4) show the sequence for PUSH and POP cycles. Note that the LIFO starts from memory address zero.



Active Write Cycle - PUSH

1. PUSH input goes active High at tsu or greater before the clock edge.
2. First clock edge samples the active PUSH input.

2a. $\overline{WR}$ goes active Low.

2b. Address increments.

3. Second clock edge; PUSH still active High.

3a. $\overline{CS}$ goes active Low.

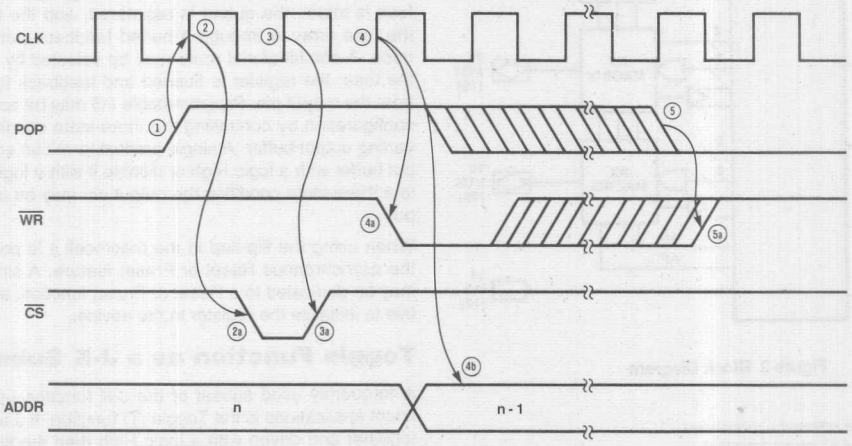
4. Third clock edge; PUSH still active High.

4a. $\overline{CS}$ goes inactive High.

5. Fourth clock edge; PUSH still active High.

5a. $\overline{WR}$ goes inactive High.

6. PUSH instruction is removed.



Active Read Cycle - POP

1. POP instruction goes active High at tsu or greater before the clock edge.
2. First clock edge samples the active POP input.

2a. $\overline{CS}$ goes active Low to select the RAM.

3. Second clock edge; POP still active High.

3a. $\overline{CS}$ goes inactive High.

4. Third clock edge; POP still active High.

4a. $\overline{WR}$ goes active Low; the memory is deselected so no Write will take place.

4b. Address lines decrement.

5. POP instruction is removed.

5a. $\overline{WR}$ goes inactive High.

Figure 4. LIFO Control Timing Diagrams

The control outputs to the RAM are $\overline{CSX}$ and $\overline{WR}$. An additional register (X0) is required to enable all states to be reached; two registers alone are not enough. From the timing waveforms it can be seen that an active write cycle is initiated after an active PUSH input has been sampled by the system clock. The $\overline{WR}$ signal then toggles, causing the output pin to go active Low. This condition is maintained until the control register X0 goes High, at the end of the PUSH cycle.

The equations after the XOR symbol in the design specification (page 2-255) cause the toggle function; when these product terms become active High a toggle operation is effected in the associated flip-flop. For example, the $\overline{CSX}$ flip-flop is in a Hold state until $\overline{WR}$ goes Low and while $\overline{X0}$ is High during a PUSH sequence; during a POP sequence, $\overline{CSX}$ holds unless both $\overline{WR}$ and $\overline{X0}$ are High. The equations are written for each flip-flop by using one product term after the XOR label to effect the change of state or toggle condition to fit the requirements of the state machine. Without the XOR gate, an Up/Down counter would require more than the eight product terms available in standard PAL devices.

Instruction Decoding

Although not shown in the design specification, it is possible for the PAL device to decode instructions for PUSH and POP. Assuming the unused input pins are converted to I0-I7, which come from an instruction register in a processor system, then decoding could be achieved in the fuse array. If an additional input $\overline{VI}$ qualifies a valid instruction and the code for PUSH is given as $\overline{I7}*\overline{I6}*\overline{I5}*\overline{I4}*I3*I2*I1*I0*\overline{VI}$, then this equation could be added to the string declaration statement of the design specification. The PAL device could decode POP operations similarly. In this way, external decoding logic may be incorporated into the PAL device, which should increase the operation speed.

2

```
TITLE           LIFO RAM CONTROLLER
PATTERN         01.
REVISION        04.
AUTHOR          CHRIS JAY.
COMPANY         MMI SANTA CLARA, CA
DATE            23 APRIL 1987
;
CHIP LIFOCONT PAL22RX8
;
;THE PAL22RX8 HAS BEEN DESIGNED TO CONTROL 32 RAM
;LOCATIONS AS A LIFO RAM CONTROLLER.  THE PAL DEVICE
;DESIGN SPECIFICATION CONSISTS OF TWO STATE MACHINES.
;ONE STATE MACHINE IS AN UP/DOWN BINARY COUNTER THAT
;DRIVES THE ADDRESS LINES OF THE RAM, AND THE SECOND
;DRIVES THE /WR AND /CS LINES OF THE RAM TO CONTROL
;WRITE AND READ OPERATIONS.  DURING A PUSH OPERATION
;DATA IS WRITTEN TO THE LIFO; THE PROCEDURE IS INC-
;REMENT ADDRESS COUNTER AND WRITE DATA.  FOR A POP
;OPERATION THE READ CYCLE IS PERFORMED FOLLOWED BY
;AN ADDRESS DECREMENT.  THE XOR GATE IN EACH MACRO-
;CELL ALLOWS INCREMENT AND DECREMENT OPERATIONS
;TO BE EFFECTED WITHOUT UTILIZING A LARGE NUMBER OF
;PRODUCT TERMS.
;
;PIN     1       2       3       4       5       6
        CLK     PUSH    POP     NC      NC      NC

;PIN     7       8       9       10      11      12
        NC      NC      NC      NC      /RST    GND

;PIN     13      14      15      16      17      18
        NC      NC      /CSX    /X0     /WR     /Q4

;PIN     19      20      21      22      23      24
        /Q3     /Q2     /Q1     /Q0     NC      VCC

GLOBAL                                  ;GLOBAL TERM
                                        ;ENABLES THE
                                        ;GLOBAL RESET
                                        ;
STRING INC '/CSX*/WR*/X0*PUSH'          ;INCREMENT LABEL
STRING DEC '/CSX*/WR* X0*POP'           ;DECREMENT LABEL
                                        ;
```

Figure 6. LIFO RAM Controller Design File

```
EQUATIONS                                   ;
                                            ;
GLOBAL.RSTF = RST                           ;RESET AT POWER ON
                                            ;
Q0      :=      Q0                          ;Q0 HOLD
        :+:     INC                         ;INC ADDRESS FOR PUSH
         +      DEC                         ;DEC ADDRESS FOR POP
                                            ;
Q1      :=      Q1                          ;Q1 HOLD
        :+:     Q0*INC                      ;INC Q1
         +     /Q0*DEC                      ;DEC Q1
                                            ;
Q2      :=      Q2                          ;Q2 HOLD
        :+:     Q0* Q1*INC                  ;INC Q2
         +     /Q0*/Q1*DEC                  ;DEC Q2
                                            ;
Q3      :=      Q3                          ;Q3 HOLD
        :+:     Q0* Q1* Q2*INC              ;INC Q3
         +     /Q0*/Q1*/Q2*DEC              ;DEC Q3
                                            ;
Q4      :=      Q4                          ;Q4 HOLD
        :+:     Q0* Q1* Q2* Q3*INC          ;INC Q4
         +     /Q0*/Q1*/Q2*/Q3*DEC          ;DEC Q4
                                            ;
CSX     :=      CSX                         ;CHIP SELECT
        :+:     WR*/X0*PUSH                 ;TO RAM DEVICE
         +     /WR*/X0*POP                  ;
                                            ;
WR      :=      WR                          ;WRITE CONTROL
        :+:    /WR*/CSX*/X0* PUSH           ;TO RAM DEVICE
         +      WR*/CSX* X0* PUSH           ;
         +     /WR*/CSX* X0* POP            ;
         +      WR*/CSX*/X0*/POP*/PUSH      ;
                                            ;
X0      :=      X0                          ;CONTROL REGISTER
        :+:     WR* CSX*/X0* PUSH           ;FOR READ-WRITE
         +     /WR*/CSX* X0*/PUSH           ;STATE MACHINE.
         +     /WR* CSX*/X0* POP            ;
         +     /WR*/CSX* X0* POP            ;
                                            ;
```

Figure 6. LIFO RAM Controller Design File (Continued)

```
SIMULATION                                  ;SIMULATION
TRACE_ON        CLK Q0 Q1 Q2 Q3 Q4          ;SECTION.
                /CSX /WR /X0 PUSH POP       ;TRACE ESSENTIAL
                RST                         ;SIGNALS.
SETF    /CLK RST /PUSH /POP                 ;RESET PAL DEVICE.
CLOCKF  CLK                                 ;CLOCK RESET TO
SETF  /RST                                  ;REGISTERS.
SETF  PUSH                                  ;ENABLE PUSH INPUT.
FOR  I := 1 TO 4 DO                         ;PERFORM ONE PUSH
BEGIN                                       ;CYCLE.
CLOCKF CLK                                  ;
END                                         ;PUSH AND POP MUST
SETF /PUSH                                  ;BE INACTIVE BETWEEN
CLOCKF CLK                                  ;CYCLES.  SET PUSH
SETF PUSH                                   ;INACTIVE.
FOR I := 1 TO 4 DO                          ;SET PUSH ACTIVE
BEGIN                                       ;AND PERFORM SECOND
CLOCKF CLK                                  ;PUSH.
END                                         ;
SETF /PUSH                                  ;SET PUSH INACTIVE
CLOCKF CLK                                  ;AND CLOCK DEVICE
SETF POP                                    ;THEN SET POP
FOR I := 1 TO 4 DO                          ;ACTIVE.  PERFORM
BEGIN                                       ;POP CYCLE.
CLOCKF CLK                                  ;
END                                         ;
SETF /POP                                   ;SET PUSH AND POP
FOR  I := 1 TO 3 DO                         ;INACTIVE.  CLOCK
BEGIN                                       ;SYSTEM FOR A PASSIVE
CLOCKF CLK                                  ;MODE.
END                                         ;
TRACE_OFF                                   ;
```

Figure 6. LIFO RAM Controller Design File (Continued)

Graphics and Image Processing Systems

Graphics systems are commonly used in most computers for displaying pictures and text. Functions of a graphics system include creation, manipulation and storage of picture databases and text. Usually, graphics systems also allow users to dynamically control a picture's content, size or colors.

Graphics systems are closely related to Image processing systems. While graphics systems manipulate images which already exist as a database, image processing systems analyze images taken from cameras by converting them to a graphical database and then processing that database. The hardware used for these systems is very similar. In fact, it is so similar that many systems offer integrated graphics and image processing functions.

In this section we will examine the possible uses of PLDs in a graphics and image processing system and their implementation. We will also determine which PLDs are suited for different applications in graphics and image processing systems. Later we will also see some design examples of PLD applications.

A simple graphics system consists of five components, as shown in Figure 1.

The first component is the graphics processor, which draws the image (referred to as "image rendering"). It also provides an interface to the main system processor (also known as the host processor). The second component is the display controller, which is an interface between the graphics processor and the graphics frame buffer. The display controller is essentially a modified DRAM controller. The third component is the graphics frame buffer, which is typically a memory that stores the image data. The data from the frame buffer is then serialized by data serializers which make up the fourth component of a graphics system. The data serializers are essentially modified shift registers. The serial data is then converted to an analog signal suitable for driving the video monitors for display. This function is performed by the fifth and the final component of the graphics system, a digital-to-analog converter (DAC).

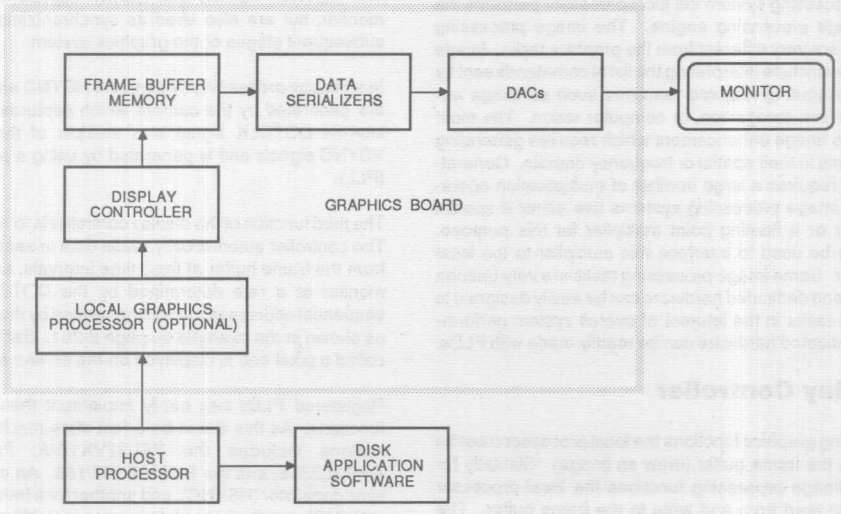
Local Graphics Processor

The graphics (or image) processor gets all the commands from the host processor. The interface between the graphics processor and the host processor is through a bus (Figure 2), where commands and data bytes are transferred. Most of the popular standard bus interfaces are used for this function. Some of the commonly used bus interfaces are:

- Multibus
- VME
- PC-bus
- Nu-bus
- Microchannel

The bus interface performs three major functions:

- Bus arbitration
- Bus control
- Address decoding



440 01

Figure 1. Components of Graphics System Hardware

There are two major types of buses, synchronous and asynchronous. For synchronous buses, the bus arbitration and control functions can be effectively performed by fast registered PLDs such as the PAL16R8/6/4, PAL20R8/6/4 and PAL22V10. For asynchronous buses, the arbitration and control functions can be provided by combinatorial PLDs, or by an asynchronous PLD (the PAL20RA10 or AmPALC29MA16). The details for designing bus interface functions are given in a separate section on page 2-325. The address decoding function is performed most effectively by a combinatorial PAL16L8, PAL20L8 or PAL22V10. Address decoding is discussed along with other combinatorial functions in the section on page 2-35.

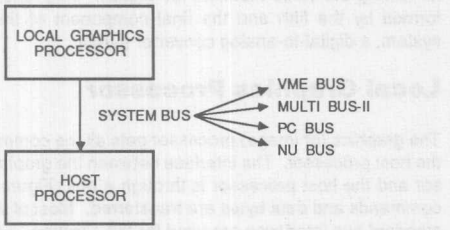


Figure 2. System Bus Interface

The local graphics processor is used to interpret the database of the graphical image ("display list"), down-loaded from the host processor. The graphics processor interprets the display list into objects with geometrical entities. It also performs coordinate transformation functions for recalculating object size as the object moves or viewer perspective changes. The last function performed by the graphics processor is the actual image rendering (drawing) functions of drawing arc/vectors, moving blocks of image data (bit-blit), and character manipulation for handling large chunks of text.

In an image processing system the local processor performs the role of an image processing engine. The image processing software tasks are very different from the graphics tasks. Image processing tasks include interpreting the list of commands sent by the host and executing required functions such as image enhancement, pattern recognition, or computer vision. The most intensive task is image enhancement which requires generating image transforms in their spatial or frequency domain. Generating transforms requires a large number of multiplication operations. Typical image processing systems use either a special microprocessor or a floating point multiplier for this purpose. PLDs can also be used to interface this multiplier to the local microprocessor. Some image processing tasks are very uniform and repetitive, and dedicated hardware can be easily designed to speed up such tasks in the interest of overall system performance. Such dedicated hardware can be readily made with PLDs.

The Display Controller

For implementing graphics functions the local processor must be able to write to the frame buffer (draw an image). Similarly for implementing image processing functions the local processor must be able to read from and write to the frame buffer. The reading and writing of frame buffer data is possible with the help of the display controller, as shown in Figure 3. The display controller is essentially a timing generator which performs three major functions:

- DRAM timing
- Monitor control signal timing
- Screen refresh

The display controller generates the read and write cycle timings as well as the refresh timing of the DRAMs. The read and write cycles are initiated by the local graphics processor. These functions are identical to a regular DRAM controller discussed in the section on memory control (page 2-179). The second function required for the display controller is the generation of monitor control signals. This function is usually integrated with the DRAM controller, although it can also be implemented separately.

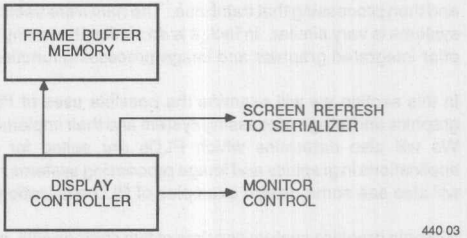


Figure 3. The Display Controller

The controller needs to generate the DOTCLK signal which determines the rate at which data is sent to the monitor. The video monitor synchronization signals HSYNC for every new line, and VSYNC for every new frame (Figure 4) are generated by dividing this DOTCLK signal. Another signal, BLANK, asserted only when the pixels are being displayed, is also generated by this controller. The HSYNC, VSYNC and BLANK signals not only drive the monitor, but are also used as synchronization signals by the subsequent stages of the graphics system.

In an image processing system, the HSYNC and VSYNC signals are generated by the camera which captures the image. The internal DOTCLK signal is a multiple of these HSYNC and VSYNC signals and is generated by using a phase locked loop (PLL).

The third function of the display controller is to refresh the screen. The controller automatically reads data in sequential addresses from the frame buffer at fixed time intervals, and sends it to the monitor at a rate determined by the DOTCLK signal. The sequential addresses are also generated by the display controller as shown in the example on page 2-261. Each chunk of data is called a pixel and is displayed on the screen as a dot.

Registered PLDs can easily implement these display control functions. As this would be a fast state machine, the range of options includes the PAL32VX10/A, PAL16/20R8/6/4, AmPAL23S8, and the PLS105/167/168. An example for a line sync generator (HSYNC), and another for a frame sync generator (VSYNC), are discussed later on pages 2-263 and 2-265 respectively.

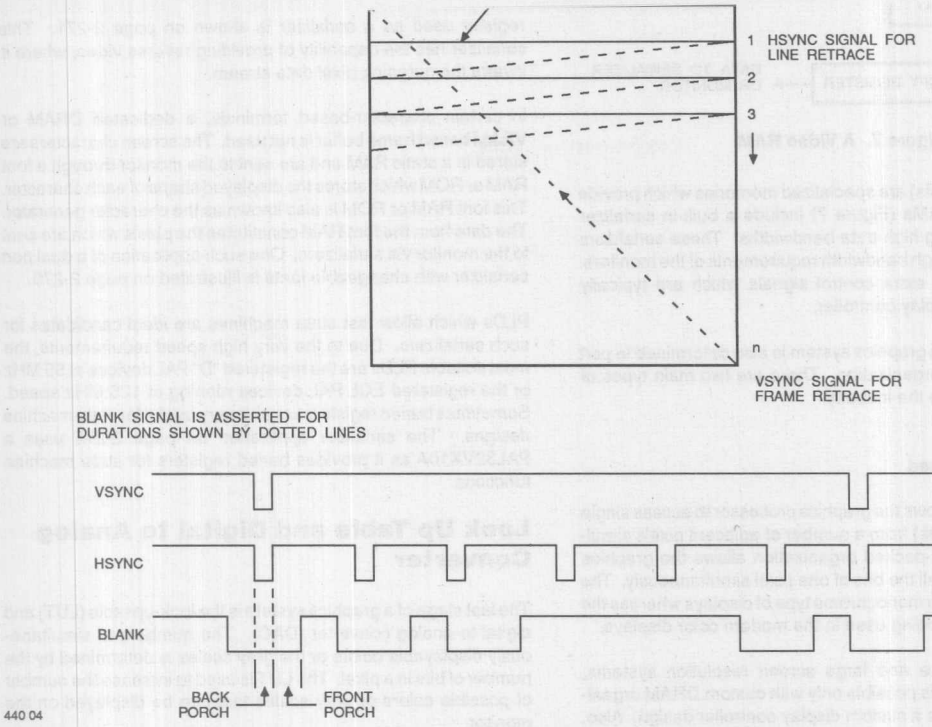


Figure 4. A Raster Scan Display Timing Control

The Frame Buffer

The frame buffer (Figure 5) is traditionally made from dynamic RAMs (DRAMs). The DRAMs provide cost effective means for storing image data. Graphics systems require two ports for reading and writing the frame buffer data. The first port is for the display update to and from the graphics processor; the second port is for the screen refresh to the monitor.

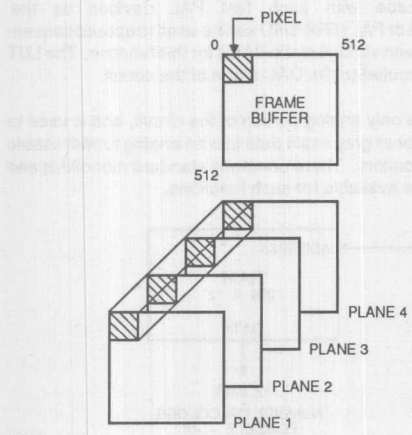


Figure 5. The Frame Buffer

The performance of a graphics system is measured by the screen update bandwidth. This is the rate at which the image data can be written to the frame buffer. The DRAMs can be organized in various ways to provide access to multiple bits of image data simultaneously, which can increase update bandwidth. Another related factor is the screen resolution, which determines the rate at which pixels have to be sent to the monitor in order to provide a flicker free display. The table in Figure 6 shows this pixel clock (DOTCLK) rate required for various screen resolutions. Varying DRAM organizations can also be used to meet this screen resolution bandwidth.

RESOLUTION	FREQUENCY	PIXEL CLOCK (DOTCLK)
512 X 512	25 MHz	40 ns
640 X 480	25 MHz	40 ns
720 X 350	50 MHz	20 ns
1000 X 1000	100 MHz	10 ns
1280 X 1000	125 MHz	8 ns

Figure 6. Various Screen Resolutions and Bandwidth Required

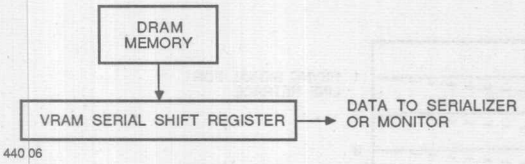


Figure 7. A Video RAM

Video DRAMs (VRAMs) are specialized memories which provide two ports. The VRAMs (Figure 7) include a built-in serializer capable of supporting high data bandwidths. These serializers are used to support high bandwidth requirements of the monitors. The VRAMs require extra control signals which are typically generated by the display controller.

The performance of a graphics system is also determined in part by the frame buffer organization. There are two main types of organizations used in the industry.

- Bit-plane mapped
- Pixel-packed mapped

Bit-plane mapping allows the graphics processor to access single bits (also called planes) from a number of adjacent pixels simultaneously. The pixel-packed organization allows the graphics processor to access all the bits of one pixel simultaneously. The former was used in the monochrome type of displays whereas the latter is increasingly being used in the modern color displays.

For high performance and large screen resolution systems, multiple pixel access is possible only with custom DRAM organizations, which require a custom display controller design. Also, different frame buffer organizations (bit-plane or pixel-packed), and the special timing requirements of VRAMs, further reduce the application of dedicated VLSI display controllers. PLDs provide an easy means of designing custom display controllers and are often used in such systems.

Serializers

The function of the serializer is to accept multiple pixels of parallel data, as sent by the display controller at slow speeds, and transform them into a serial pixel data stream. The serializer is required even for VRAMs, which can only support 25 MHz bandwidths. The serializer speed requirements are based on monitor resolution and are shown in Figure 5.

The serializer is synchronized by the monitor control signals HSYNC, VSYNC and BLANK, which are generated by the display controller. A serializer is essentially a shift register, similar to the one discussed earlier on page 2-90. Certain modifications are added to accommodate the specific design requirements of a graphics system. These include synchronization with the display control signals.

Many standard video serializers (679501, 679502, and Am8177) are available, which can be used for standard DRAM organizations. For non-standard organizations, custom serializers can be made with PLDs. Sometimes graphics-related functions, which are not available in standard serializers, can also be added to such custom serializers. For example, a ZOOM function can be easily implemented in a custom serializer by having it repeat the same pixel data two or more times. An example of a seven bit shift register used as a serializer is shown on page 2-271. This serializer has the capability of providing reverse video, where it inverts the outgoing pixel data stream.

In certain character-based terminals, a dedicated DRAM or VRAM based frame buffer is not used. The screen characters are stored in a static RAM and are sent to the monitor through a font RAM or ROM which stores the displayed shape of each character. This font RAM or ROM is also known as the character generator. The data from the font RAM constitutes the pixels which are sent to the monitor via serializers. One such application of a dual port serializer with changeable fonts is illustrated on page 2-275.

PLDs which allow fast state machines are ideal candidates for such serializers. Due to the very high speed requirements, the most suitable PLDs are the registered "D" PAL devices at 55 MHz or the registered ECL PAL devices running at 125 MHz speed. Sometimes buried registers are also very useful for state machine designs. The serializer application on page 2-275 uses a PAL32VX10A as it provides buried registers for state machine functions.

Look Up Table and Digital to Analog Converter

The last stage of a graphics system is the look-up table (LUT) and digital-to-analog converter (DAC). The numbers of simultaneously displayable colors or the gray scales is determined by the number of bits in a pixel. The LUT is used to increase the number of possible colors or gray scales that can be displayed on the monitor.

The LUT (Figure 8) is a fast RAM addressed by the pixels from the serializers. The data stored in the RAMs is used as the color or gray scale value. Since the data speeds can be very fast (as little as 8 ns for a 125 MHz display) and the static RAM access times are relatively slow, multiple static RAMs are used to store identical copies of look up tables. Different static RAMs are addressed for sequential pixels in order to maintain a fast overall data rate. Controllers made with such fast PAL devices as the PAL10H20EV8 or PAL16R8/6/4D can be used to spread sequential pixels between various static RAMs for this function. The LUT output is then routed to the DAC stage of the circuit.

The DAC is the only analog portion of the circuit, and is used to convert the color or gray scale data into an analog format usable by the video monitors. There are many standard monolithic and hybrid solutions available for such functions.

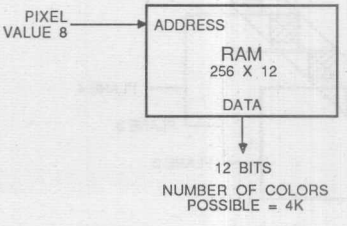


Figure 8. The Lookup Table

Small System Video Controller

This small system video controller is designed for an 80 character wide terminal display of 24 rows (Figure 1). The display resolution is 640 X 400 pixels, with a character box of 8 X 16 pixels. The character font used is a dot matrix of 7 X 9 pixels which is stored in a character PROM. The first seven horizontal pixels display the character while the eighth pixel provides the horizontal character separation. Similarly the first nine lines display a visible character while the remaining seven are blank to provide line separation. The frame buffer is an SRAM which stores character ASCII codes of 80 X 24 characters and can be addressed by the system processor for update or by the display controller.

The display controller provides all the necessary timing and addressing for supporting this display. Synchronized to the monitor timing signals, it generates the row and column addresses (Figure 2), of the character being scanned. The row and column addresses access the character ASCII code from the frame buffer at the instant the character is displayed. This ASCII code addresses the character PROM which stores the 7 X 9 dot matrix font. Suppose the letter "A" is to be displayed (Figure 3). The contents of the character PROM pertaining to the current character row and column are read and loaded into the video shift register to be clocked out as a video signal. The video shift register must be seven-bits long, and the entire character is

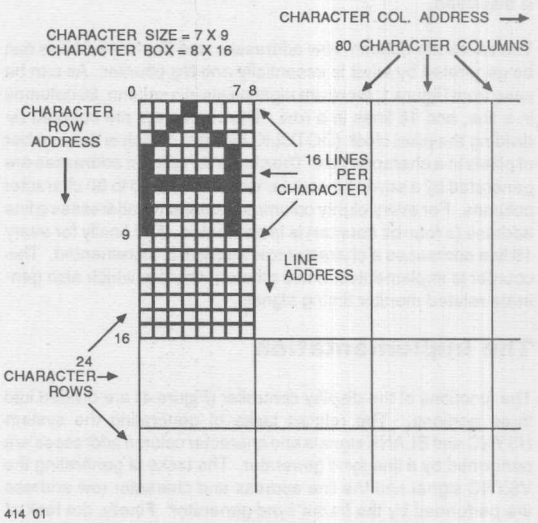


Figure 1. A Character Display

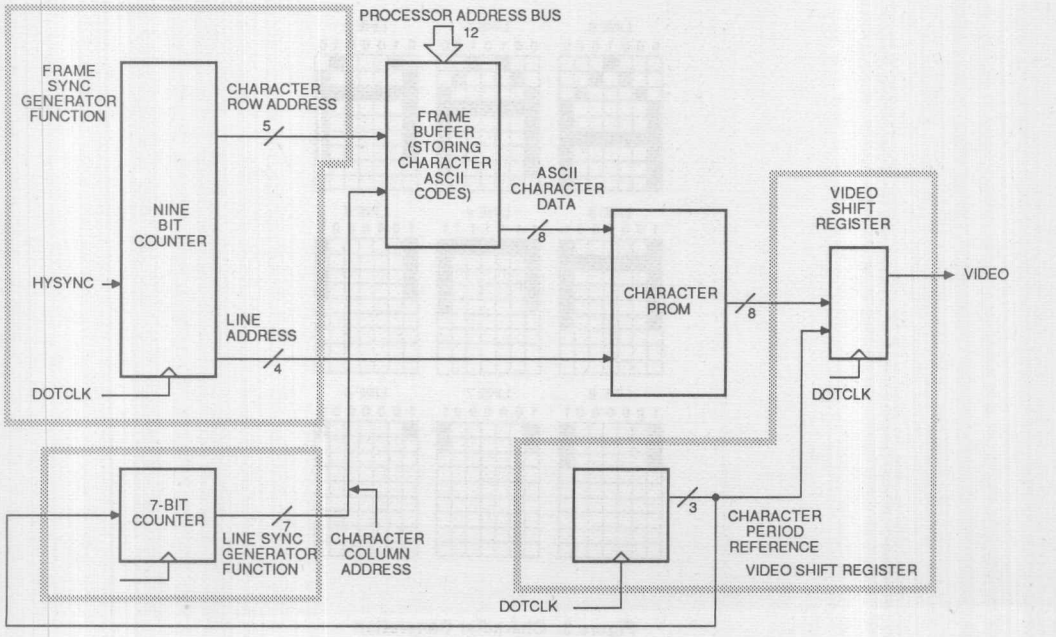


Figure 2. Functional Block Diagram

displayed over nine lines. To display a character 'A', the ASCII code of '41' hexadecimal selecting the letter is read from a frame buffer location. This character code will point to the base location in the PROM. This base location holds the first line of a table of nine locations which hold the dot information for each scan line. The line address counter first selects the R0 line address, then the next line address R1 and then R2 to R8, until the entire character is displayed.

The character column, row addresses and the line address can be generated by what is essentially one big counter. As can be seen from Figure 1, there are eight pixels in a column, 80 columns in a line, and 16 lines in a row. The characters are counted by dividing the pixel clock (DOTCLK) by eight, which is the number of pixels in a character box. The character column addresses are generated by a seven bit counter which counts up to 80 character columns. For every eighty columns of character addresses a line address (a four-bit counter) is incremented. And finally for every 16 line addresses a character row address is incremented. This counter is implemented in two different devices which also generate related monitor timing signals.

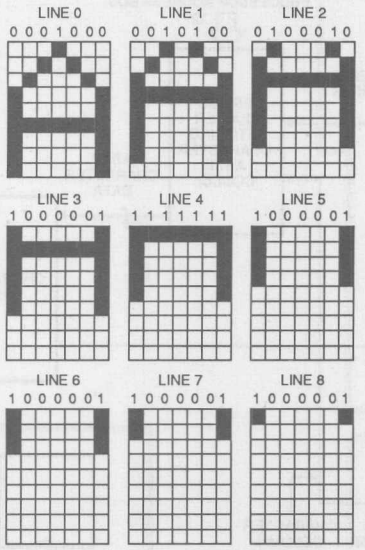
The Implementation

The functions of the display controller (Figure 4) are divided into three portions. The related tasks of generating the system HSYNC and BLANK signals and character column addresses are performed by a line sync generator. The tasks of generating the VSYNC signal and the line address and character row address are performed by the frame sync generator. Finally, the task of generating character period reference from the DOTCLK has been conveniently combined with the video shift register, which also uses this reference to shift out appropriate font pixels. The character period reference is also used by line sync and frame sync generators to count number of characters.

The sequential frame buffer column address MA0–MA6 for displaying these 80 characters per row, is generated by the line sync generator (IC2 in Figure 4). The line sync generator is a counter clocked by DOTCLK with a count enable function under the control of a character period reference. This enables it to count the number of character periods. The line sync generator also generates HSYNC and BLANK signals at the end of each line.

The character display is divided into 24 rows of 16 lines each. The four-bit line address (R0–R3) is used by the character PROM to generate the appropriate bits of font for the different lines in a character row. The line address to the character PROM is generated by the frame sync generator (IC3), using the BLANK signal as a new line reference. For every new line (BLANK signal) the line address counter is incremented. The frame sync generator also generates the five-bit row address (A4–A8) for the 24 different rows of characters stored in the frame buffer. This address is generated by a counter which is incremented once for every 16 counts of the line address counter.

The final component is the video shift register (IC1) which serializes the character PROM seven-bit character line data. This function is accomplished by using a three-bit counter which divides the DOTCLK frequency by eight to generate the character period reference.



414 02

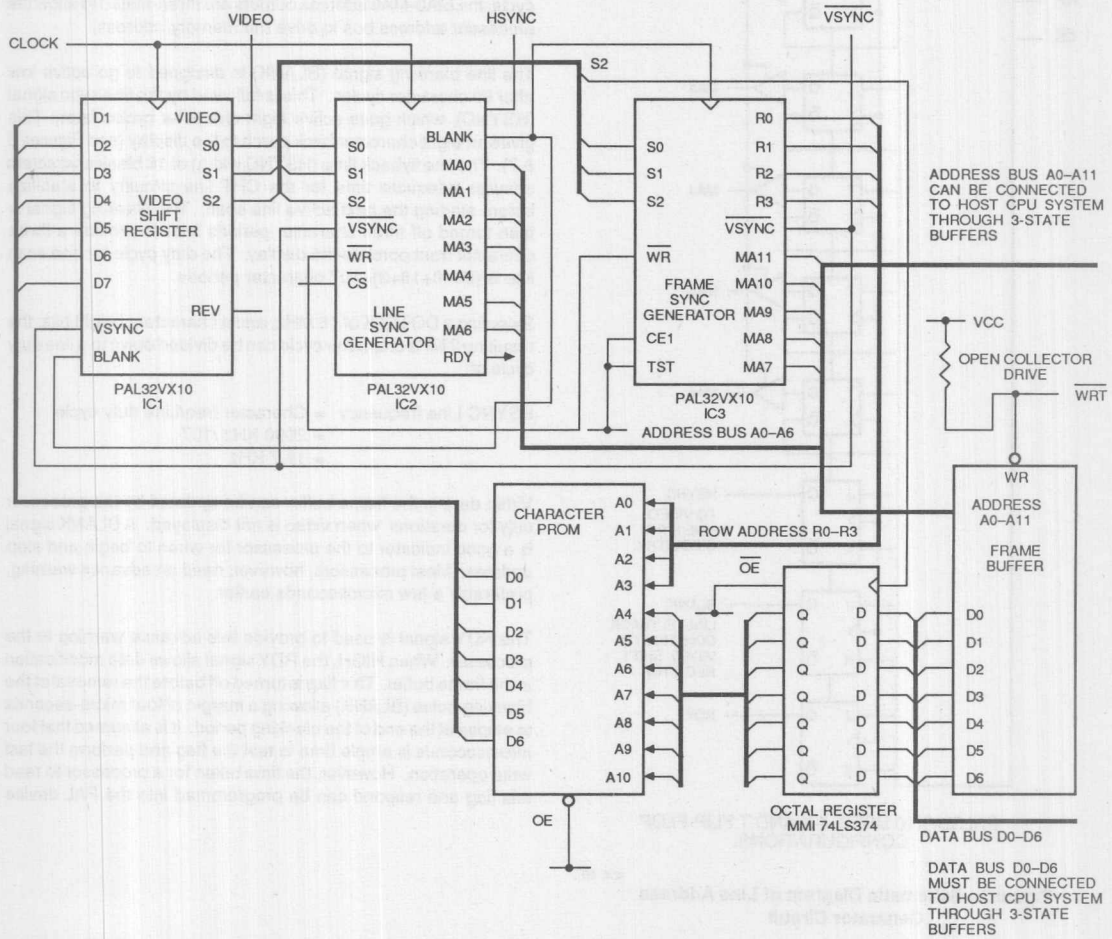
Figure 3. Character Generation

Line Sync Generator

The PAL32VX10 used for line sync generation (Figure 5) has a versatile macrocell structure capable of transforming a basic D-type flip-flop into a more complex T-type flip-flop which efficiently implements counter designs. The seven-bit character column address counter uses seven of these T-type flip-flops. The line sync (HSYNC) signal is generated based on the value of the counter. The HSYNC signal initiates line flyback which separates the many lines of which a visual display is composed. The device also generates the blanking signal for monitor control. This design has been developed into state machine logic contained in the PAL32VX10.

Addressing the Frame Buffer

The frame buffer address lines MA0-MA6 are shared by a host processor which updates the memory with new characters. These addresses are generated by a binary character period counter. The only system clock conveniently available is the DOTCLK. Since the counter has to keep track of character periods. One easy way is to clock the device with DOTCLK, but enable count only when all three bits of the pixel counter S2, S1 and S0 are high. These three bits high indicate a character boundary; this will effectively result in clocking the counter every character period.



414 04

Figure 4. Small System Video Controller

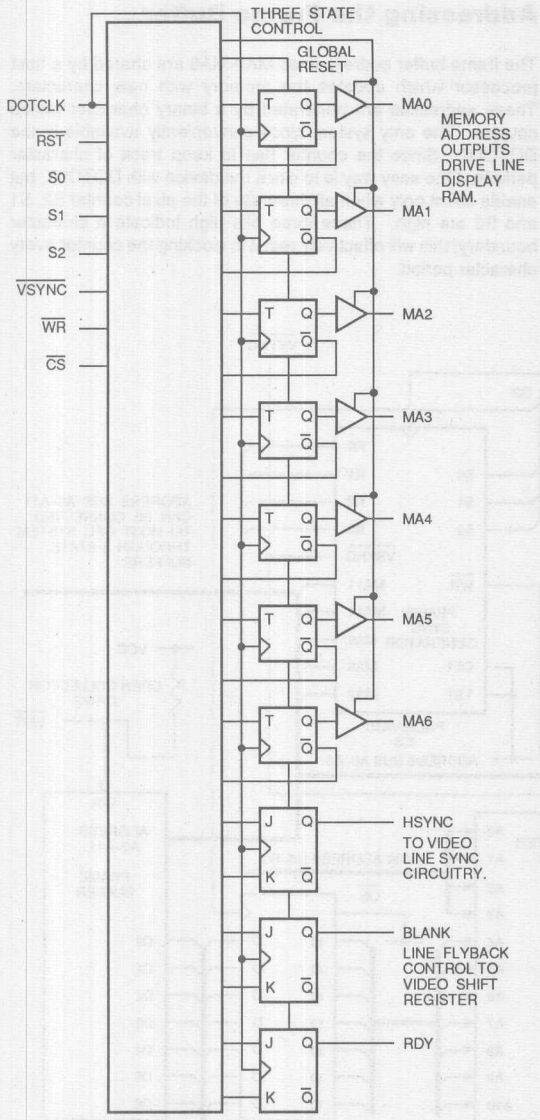


Figure 5. Schematic Diagram of Line Address Generator Circuit

The count is extended up to 107 to provide a timing reference to generate the monitor control signals (HSYNC and BLANK) also. Seven registers in the PAL32VX10 provide this count. The registers are programmed as T-type flip-flops (Figure 5) for optimizing the number of product terms used. For each scan line after reaching 107 the T-type counters are reset to provide the start of the next active line. Of the ten registered cells in the PAL32VX10, nine registers are used for addressing memory (MA0-MA6), line sync generation (HSYNC) and blanking control (BLANK).

Since the processor can not read or write to the frame buffer at the same time as the display controller, the address lines (MA0-MA6) of the display controller are three-stated during a processor cycle. When processor signal /WR is asserted LOW, indicating a write cycle, the MA0-MA6 address outputs are three-stated to allow the processor address bus to drive the memory address.

The line blanking signal (BLANK) is designed to go active low after 80 character cycles. This is followed by the line sync signal (HSYNC), which goes active eight character cycles later. This gives an eight character back porch to the display (see Figures 6 & 7). The line flyback time (HSYNC width) of 16 blank characters ensures adequate time for the CRT line circuitry to stabilize before starting the next active line scan. The blanking signal is then turned off three character periods later to provide a three character front porch to the display. The duty cycle for one scan line is (80+8+16+3)=107 character periods.

Selecting a DOTCLK of 16 MHz, and a character of eight bits, the resulting 2 MHz character cycle can be divided down to a line duty cycle of:

HSYNC Line frequency = Character freq/Line duty cycle
= 2000 KHz /107
= 18.7 KHz

Video data in the frame buffer can be updated by the processor only for durations when video is not displayed. A BLANK signal is a good indicator to the processor for when to begin and stop updates. Most processors, however, need an advance warning, preferably a few microseconds earlier.

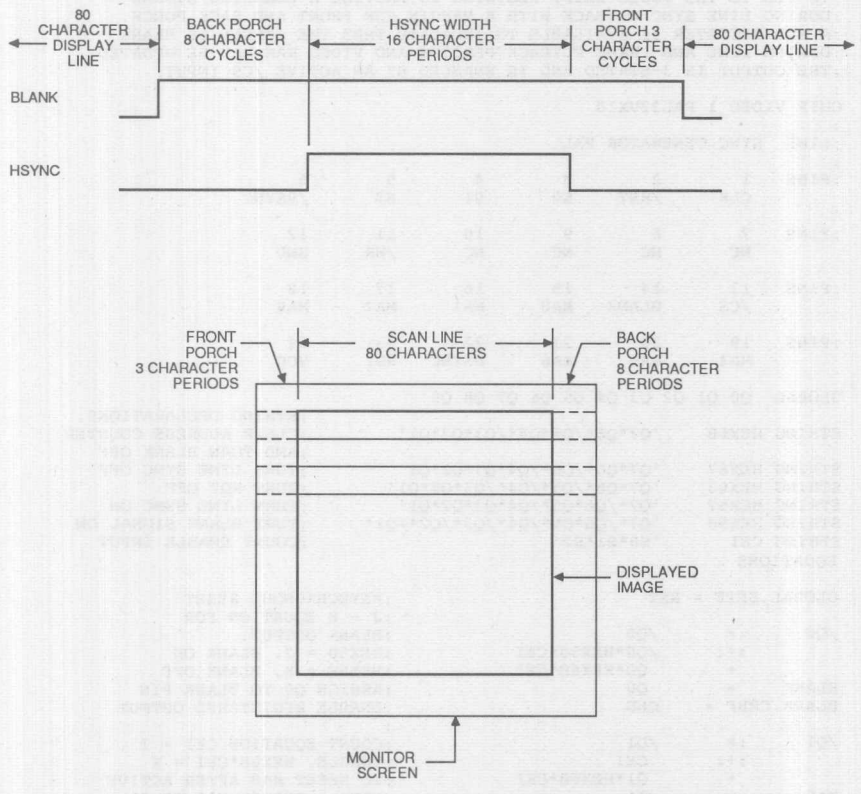
The RDY signal is used to provide this advance warning to the processor. When HIGH, the RDY signal allows data modification in the frame buffer. This flag is turned off before the removal of the blanking pulse (BLANK) allowing a margin of four micro-seconds to elapse at the end of the blanking period. It is assumed that four microseconds is ample time to test the flag and perform the last write operation. However, the time taken for a processor to read this flag and respond can be programmed into the PAL device

depending upon the software requirements of the system. The RDY signal is also set during vertical retrace when all processor updates are allowed and no display controller accesses are required. The design uses the system VSYNC for accomplishing this.

The RDY signal is also disabled unless it is being read by the processor. This is done because the RDY signal is usually directly connected to the processor data bus which is used to carry data from different sources (drivers). The disabling is done using the $\overline{CS}$ chip-select signal asserted by the processor. If $\overline{CS}$ is driven LOW, the RDY output buffer is enabled and the RDY flag may be read by the processor.

Frame Sync Generator

Another PAL32VX10 (Figures 8 & 9) is programmed to perform the function of a frame sync generator (VSYNC), with the ability to access the character PROM and the frame buffer. The VSYNC signal controls frame flyback for the CRT controller. Each frame of displayed data is terminated, and the electron beam in the CRT performs a retrace operation prior to the display of the next frame. Clocked by the system BLANK signal, it uses a counter to generate the row address for the character being displayed. It also generates the line address to access the dot pattern for each scan line, stored in character PROM.



414 06

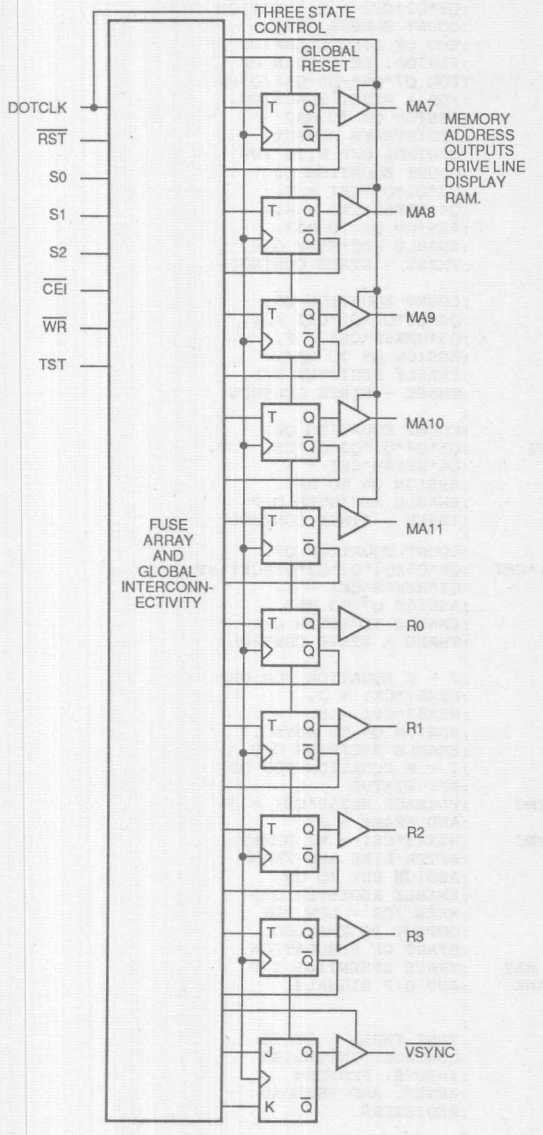
Figure 6. Blanking and Sync. Signals.

```
AUTHOR          CHRIS JAY.
COMPANY         MMI SANTA CLARA, CA.
DATE            20TH AUGUST 1986.
;
;THE PAL32VX10 HAS BEEN DESIGNED AS A LINE SYNC GENERATOR FOR A
;THREE PAL SOLUTION TO A VIDEO DISPLAY CONTROLLER.  THE DEVICE
;HAS BEEN PROGRAMMED TO GENERATE ADDRESSING TO ACCESS A CHARACTER
;RAM MA0 - MA6 ARE CONFIGURED TO ACCESS ASCCII CHARACTERS STORED IN
;THE RAM FOR AN EIGHTY CHARACTER WIDE DISPLAY. THESE ADDRESS OUTPUTS
;MAY BE PUT INTO A THREE STATE CONDITION IF THE HOST PROCESSOR
;REQUIRES TO CHANGE THE CONTENTS OF THE DISPLAYED MEMORY. WITH
;THE /WR INPUT ACTIVE LOW THE HOST PROCESSOR MAY DRIVE THE
;ADDRESS INPUTS TO ACCESS AND MODIFY THE CONTENTS OF THE VIDEO RAM.
;THE INTERFACE SIGNALS TO THE VIDEO SHIFT REGISTER (PAL DESIGN 01
;OF THE THREE PAL DESIGNS) ARE S0,S1,S2, AND BLANK. WHEN S0*S1*S2
;ARE HIGH THE ADDRESS OUTPUTS MA0 - MA6 INCREMENT SYNCHRONOUSLY
;AS A RESULT OF THE DOT CLOCK. S0,S1 AND S2 ARE STATE MACHINE
;OUTPUTS FROM THE VIDEO SHIFT REGISTER THAT CONTROL THE LOADING
;AND SHIFTING OF PIXEL DATA. THE BLANK OUTPUT IS ACTIVE HIGH AND
;IS FED TO THE VIDEO SHIFT REGISTER TO PROVIDE A BLANKING SIGNAL
;DURING LINE SYNC FLYBACK WITH A MARGIN FOR FRONT AND BACK PORCH.
;A RDY REGISTER IS AVAILABLE TO INDICATE THAT THE SCREEN IS BLANK
;DURING LINE AND FRAME FLYBACK PERIODS AND VIDEO RAM MAY BE UPDATED.
;THE OUTPUT IS 3-STATED AND IS ENABLED BY AN ACTIVE /CS INPUT.
;
CHIP VIDEO_1 PAL32VX10
;
;LINE  SYNC GENERATOR PAL
;
;PINS   1       2       3       4       5       6
        CLK     /RST    S0      S1      S2      /VSYNC

;PINS   7       8       9       10      11      12
        NC      NC      NC      NC      /WR     GND

;PINS   13      14      15      16      17      18
        /CS     BLANK   MA0     MA1     MA2     MA3

;PINS   19      20      21      22      23      24
        MA4     MA5     MA6     HSYNC   RDY     VCC

GLOBAL  Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9
                                                ;STRING DECLARATIONS.
STRING HEX6B    'Q7*Q6*/Q5*Q4*/Q3*Q2*Q1'        ;CLEAR ADDRESS COUNTER
                                                ;AND TURN BLANK OFF
STRING HEX67    'Q7*Q6*/Q5*/Q4*Q3*Q2*Q1'        ;TURN LINE SYNC OFF
STRING HEX63    'Q7*Q6*/Q5*/Q4*/Q3*Q2*Q1'       ;TURN RDY OFF
STRING HEX57    'Q7*/Q6*Q5*/Q4*Q3*Q2*Q1'        ;TURN LINE SYNC ON
STRING HEX50    'Q7*/Q6*Q5*/Q4*/Q3*/Q2*/Q1'     ;TURN BLANK SIGNAL ON
STRING CEI      'S0*S1*S2'                      ;COUNT ENABLE INPUT
EQUATIONS

GLOBAL.SETF = RST                               ;ASYNCHRONOUS RESET
                                                ;J - K EQUATION FOR
/Q0     :=      /Q0                             ;BLANK OUTPUT.
        :+:     /Q0*HEX50*CEI                   ;HEX50 = J. BLANK ON
         +       Q0*HEX6B*CEI                   ;HEX6B = K. BLANK OFF
BLANK   =       Q0                              ;ASSIGN Q0 TO BLANK PIN
BLANK.CMBF =    GND                             ;ENABLE REGISTERED OUTPUT
                                                ;
/Q1     :=      /Q1                             ;COUNT EQUATION CEI = T
        :+:      CEI                            ;TOGGLE. HEX6B*CEI = K
         +       Q1*HEX6B*CEI                   ;TO RESET MA0 AFTER ACTIVE
MA0     =       Q1                              ;LINE.  ASSIGN MA0 TO Q1
MA0.CMBF =      GND                             ;REGISTER.
MA0.TRST =       /WR                            ;ENABLE THREE-STATE
                                                ;DURING MPU/RAM ACCESS.
/Q2     :=      /Q2                             ;COUNT EQUATION Q2
        :+:      Q1*CEI                         ;Q1*CEI = T FUNCTION
         +       Q2*HEX6B*CEI                   ;HEX6B*CEI = K
MA1     =       Q2                              ;ENABLE Q2 REGISTER TO
MA1.CMBF =      GND                             ;MA1 OUTPUT. CONTROL
MA1.TRST =       /WR                            ;THREE-STATE DURING
                                                ;MPU/RAM ACCESS.
```

Figure 7. Design File of Line Sync Generator

```
                +       Q2*Q1*CEI*/Q4           ;PERIOD. DE MORGAN OF
                +       Q2*Q1*CEI*Q3            ;EQU Q7*Q6*/Q5*Q4*/Q3=K
MA2             =       Q3                      ;/Q9 = RESET FUNCTION.
MA2.CMBF        =       GND                     ;ASSIGN Q3 TO MA2
MA2.TRST        =       /WR                     ;REGISTERED OUTPUT.
                                                ;CONTROL O/P WITH /WR
/Q4             :=      /Q4                     ;COUNT EQUATION Q4
                :+:     Q3*Q2*Q1*CEI            ;Q3*Q2*Q1*CEI = T.
                +       Q4*HEX6B*CEI            ;Q4*HEX6B*CEI = K.
MA3             =       Q4                      ;ASSIGN Q4 TO MA3.
MA3.CMBF        =       GND                     ;ENABLE REGISTER O/P.
MA3.TRST        =       /WR                     ;THREE - STATE CONTROL.
                                                ;
/Q5             :=      /Q5                     ;COUNT EQUATION Q5.
                :+:     Q4*Q3*Q2*Q1*CEI         ;Q4*Q3*Q2*Q1*CEI = T.
                +       Q5*HEX6B*CEI            ;Q5*HEX6B*CEI = K.
MA4             =       Q5                      ;ASSIGN Q5 TO MA4.
MA4.CMBF        =       GND                     ;ENABLE REGISTER O/P.
MA4.TRST        =       /WR                     ;THREE - STATE CONTROL.
                                                ;
/Q6             :=      /Q6                     ;COUNT EQUATION Q6.
                :+:     Q5*Q4*Q3*Q2*Q1*CEI      ;Q5*Q4*Q3*Q2*Q1*CEI = T.
                +       Q6*HEX6B*CEI            ;Q6*HEX6B*CEI = K.
MA5             =       Q6                      ;ASSIGN Q6 TO MA5.
MA5.CMBF        =       GND                     ;ENABLE REGISTER O/P
MA5.TRST        =       /WR                     ;THREE - STATE CONTROL
                                                ;
/Q7             :=      /Q7                     ;COUNT EQUATION Q7.
                :+:     Q6*Q5*Q4*Q3*Q2*Q1*CEI   ;Q6*Q5*Q4*Q3*Q2*Q1*CEI =T.
                +       Q7*HEX6B*CEI            ;Q7*HEX6B*CEI = K.
MA6             =       Q7                      ;ASSIGN Q7 TO MA6.
MA6.CMBF        =       GND                     ;ENABLE REGISTER O/P
MA6.TRST        =       /WR                     ;THREE - STATE CONTROL
                                                ;
/Q8             :=      /Q8                     ;J - K EQUATION FOR Q8
                :+:     /Q8*HEX57*CEI           ;HEX57*CEI = J.
                +       Q8*HEX67*CEI            ;HEX67*CEI = K.
/HSYNC          =       /Q8                     ;ASSIGN Q8 TO HSYNC.
/HSYNC.CMBF     =       GND                     ;ENABLE REGISTER O/P
                                                ;J - K EQUATION FOR Q9
/Q9             :=      /Q9                     ;RDY STATUS
                :+:     /Q9*HEX50*CEI*/VSYNC    ;FLYBACK.HEX50*CEI = J1
                +       /Q9*CEI*VSYNC           ;AND FRAME = J2
                +       Q9*HEX63*CEI*/VSYNC     ;HEX63*CEI = K1 RESET
                                                ;AFTER LINE AND FRAME
RDY             =       Q9                      ;ASSIGN RDY TO Q9
RDY.CMBF        =       GND                     ;ENABLE REGISTER O/P
RDY.TRST        =       CS                      ;WHEN /CS = LOW RDY
                                                ;OUTPUT IS ENABLED.
SIMULATION                                      ;START OF SIMULATION.
TRACE_ON CLK RST S0 S1 S2 MA0 MA1 MA2           ;TRACE ESSENTIAL I/P
         MA3 MA4 MA5 MA6 HSYNC BLANK            ;AND O/P SIGNALS.
         RDY /VSYNC                             ;
SETF     WR /VSYNC                              ;
                                                ;TEST THREE - STATE
SETF     /CLK RST S0 S1 S2 /WR CS               ;CONTROL. INITALISE
CLOCKF   CLK                                    ;INPUTS. PERFORM
SETF     /RST                                   ;RESET. AND PRELOAD
PRLDF    /Q0 /Q1 /Q2 /Q3 Q4                     ;REGISTERS.
         /Q5 /Q6 Q7 /Q8 /Q9                     ;
FOR I := 1 TO 60 DO                             ;APPLY 60 CLOCK PULSES
BEGIN                                           ;TO TRACE THE MEMORY
CLOCKF  CLK                                     ;ADDRESS COUNTER, HSYNC,
END                                             ;BLANK AND CEO.
SETF VSYNC
FOR I := 1 TO 8 DO                              ;TEST THAT AN ACTIVE
BEGIN CLOCKF CLK                                ;FRAME WILL SET THE
END                                             ;RDY FLAG.
SETF     /S0                                    ;DISABLE COUNTER BY
CLOCKF   CLK                                    ;SETTING S0,S1 AND S2
SETF     /S1                                    ;LOW.
CLOCKF   CLK                                    ;
SETF     /S2                                    ;
CLOCKF                                          ;
TRACE_OFF                                       ;END OF SIMULATION.
```

Figure 7. Design File of Line Sync Generator (Cont'd.)



PAL32VX10 USES J–K AND T FLIP-FLOP CONFIGURATIONS.

414 07

Figure 8. Schematic of Frame Sync Generator

The PAL device is programmed as a counter. The least significant four bits count through sixteen line addresses R0–R3 for each character row. The higher significant bits of the counter track the 24 rows of characters. The frame sync pulse is generated on the 25th character row and frame flyback is turned on for sixteen line count periods, which is the duration of the twenty-fifth row. With 24 character rows, each having sixteen lines, the system gave 384 visible lines. Since the frame flyback occurs over sixteen lines, the total number of lines in the system is (384+16)= 400. The frame repetition rate is derived by dividing the line frequency by the total number of lines in the system.

Frame repetition rate = $18.7 \times 10^{+3}$ HZ/400.....(1).
= 46.75 Hz

The video terminal used for testing the final design functioned best at this frame repetition rate. If desired, other frame repetition rates can also be selected. For example a thirteen character row with nine rows active and the same 7 X 9 character box would give a 57 Hz frame rate. Most video terminals have a synchronizing circuit that determines the choice of frame frequency. This is similar to a phase-locked loop, providing a window of acceptable frame oscillator frequencies to which the display will be locked. Outside of this capture range, the display will not be synchronized, and the result will look like a rolling unstable picture.

Signal Assignments

The /VSYNC output is an active LOW signal. When displaying data this output is driven HIGH. As with the line sync generator design, the /WR input is driven from the host processor system. When LOW, the video memory address outputs MA7–MA11 are disabled. The host system may enable its output buffers and drive the frame buffer's address inputs directly to modify video data.

For normal counting, the $\overline{CEI}$ input should be driven LOW. A TST input was provided to minimize the number of test vectors required to test the device. This pin is driven LOW for normal operation.

```
TITLE           FRAME SYNC GENERATOR.
PATTERN         02
REVISION        02.
AUTHOR          CHRIS JAY.
COMPANY         MMI SANTA CLARA, CA.
DATE            20TH AUGUST 1986.
;
;
CHIP VIDEO_1 PAL32VX10
;
;FRAME  SYNC GENERATOR PAL
;
;THE PAL 32VX10 IS THE THIRD OF A THREE PAL SYSTEM DESIGNED
;AS A SIMPLE VIDEO CONTROLLER. THE DEVICE GENERATES ADDRESSING
;FOR A CHARACTER MEMORY AND CPU RAM. ALSO A FRAME SYNC PULSE
;/VSYNC IS USED TO PROVIDE FRAME FLYBACK AT THE END OF ONE
;COMPLETE FRAME. THERE ARE FOUR ROW OUTPUTS FROM THE PAL WHICH
;ADDRESS THE CHARACTER MEMORY. THESE OUTPUTS ARE DRIVEN FROM
;AN INTERNAL COUNTER WHICH IS INCREMENTED AFTER EACH LINE
;SCAN. THE R0,R1,R2 AND R3 INCREMENT AS A SIXTEEN BIT COUNTER
;ADDRESSING CHARACTER MEMORY. MA7 - MA11 ADDRESS VIDEO MEMORY
;ACCESSING THE ASCII CODES OF THE CHARACTER TO BE DISPLAYED
;ON EACH ROW OF THE VIDEO DISPLAY. THE COUNTER MA7 - MA11
;IS INCREMENTED AFTER ONE COMPLETE CHARACTER SCAN THAT IS
;SIXTEEN LINES. THE /WR INPUT CAN BE DRIVEN BY THE HOST CPU
;TO 3 - STATE MA7 - MA11 OUPUTS AND DYNAMICALLY CHANGE THE
;CONTENTS OF THE VIDEO RAM. THIS INPUT CAN BE MAPPED INTO
;HOST PROCESSOR'S MEMORY ADDRESS RANGE. UPDATING RAM SHOULD
;BE DONE BY THE HOST DURING LINE AR FRAME FLYBACK. THE PAL
;DEVICE MAY BE RESET AT POWER ON AND /CEI SHOULD BE WIRED LOW
;TO ENABLE THE ADDRESS COUNTERS. TST IS HARDWIRED LOW, THIS
;PIN WAS USED AS A TEST INPUT PULLED HIGH TO DISABLE SOME OF
;LOWER ORDER COUNTER REGISTERS WHEN GENERATING TEST VECTORS
;OF THE TEST VECTORS. THIS AVOIDED GENERATING UNECESSARY
;REPETATIVE LOWER ORDER COUNTS WHEN TESTING THE COUNT
;EQUATIONS FOR HIGHER ORDER REGISTERS IN THE COUNTER
;CHAIN.
;
;PINS   1       2       3       4       5       6
        CLK     /RST    /CEI    NC      NC      NC

;PINS   7       8       9       10      11      12
        NC      NC      NC      /WR     TST     GND

;PINS   13      14      15      16      17      18
        NC      MA7     MA8     MA9     MA10    MA11

;PINS   19      20      21      22      23      24
        /VSYNC  R0      R1      R2      R3      VCC

GLOBAL  Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9
;
;STRING VARIABLE DECLARATIONS FOR A NON-INTERLACED SYSTEM.
;
STRING HEX18NI  'Q9*Q8*Q7*Q6*Q4*Q3*/Q2*/Q1*/Q0'         ;SET VSYNC
STRING HEX19NI  'Q9*Q8*Q7*Q6*Q4*Q3*/Q2*/Q1*Q0'          ;CLEAR VSYNC
STRING ROWNI    'Q6*Q7*Q8*Q9'                           ;ROW COMPLETE
;
;

EQUATIONS

GLOBAL.SETF = RST                       ;RESET AT POWER UP
/Q0     :=      /Q0                     ;COUNT EQUATION.
        :+:      CEI*ROWNI              ;TOGGLE WHEN CEI*ROWNI TRUE
         +       CEI*HEX19NI            ;RESET AFTER FRAME SCAN.
MA7     =        Q0                     ;ENABLE MA7 OUTPUT
MA7.CMBF =       GND                    ;SET REGISTER MODE
MA7.TRST =      /WR                     ;THREE - STATE O/P WHEN /WR
                                        ;ACTIVE.
/Q1     :=      /Q1                     ;COUNT EQUATION Q1
        :+:      Q0*CEI*ROWNI*/Q5       ;ENABLE TOGGLE
MA8     =        Q1                     ;ENABLE MA8 OUTPUT.
MA8.CMBF =       GND                    ;SET REGISTER MODE.
MA8.TRST =      /WR                     ;THREE - STATE O/P WHEN /WR
                                        ;ACTIVE.
```

Figure 9. Design File of Frame Sync Generator

```
                                                    ;RESET AFTER FRAME SCAN
MA9       =       Q2                                ;ENABLE MA9 OUTPUT.
MA9.CMBF  =       GND                               ;SET REGISTER MODE.
MA9.TRST  =      /WR                                ;THREE - STATE O/P WHEN /WR
                                                    ;ACTIVE.
/Q3       :=     /Q3                                ;COUNT EQUATION Q3
          :+:     Q2*Q1*Q0*CEI*ROWNI                ;ENABLE TOGGLE
           +      Q3*HEX19NI*CEI                    ;RESET AFTER FRAME SCAN
MA10      =       Q3                                ;ENABLE MA10 OUTPUT.
MA10.CMBF =       GND                               ;SET REGISTER MODE
MA10.TRST =      /WR                                ;THREE - STATE O/P WHEN /WR
                                                    ;ACTIVE.
/Q4       :=     /Q4                                ;COUNT EQUATION Q4
          :+:     Q3*Q2*Q1*Q0*CEI*ROWNI             ;ENABLE TOGGLE
           +      Q4*HEX19NI*CEI                    ;RESET AFTER FRAME SCAN
MA11      =       Q4                                ;ENABLE MA11 OUTPUT
MA11.CMBF =       GND                               ;SET REGISTER MODE
MA11.TRST =      /WR                                ;THREE - STATE O/P WHEN /WR
                                                    ;ACTIVE.
/Q5       :=     /Q5                                ;J - K VSYNC O/P
          :+:    /Q5*HEX18NI*CEI*ROWNI              ;J EQUATION TO SET VSYNC
           +      Q5*HEX19NI*CEI*ROWNI              ;K EQUATION TO RESET VSYNC
/VSYNC    =      /Q5                                ;ENABLE Q5 TO VSYNC.
/VSYNC.CMBF =     GND                               ;SET REGISTER MODE
                                                    ;
/Q6       :=     /Q6                                ;COUNT EQUATION FOR Q6
          :+:     CEI                               ;TOGGLE
R0        =       Q6                                ;FRAME SYNC. ENABLE Q6
R0.CMBF   =       GND                               ;REGISTER OUTPUT FOR
                                                    ;ROW 0.
/Q7       :=     /Q7                                ;COUNT EQUATION FOR Q7
          :+:     Q6*CEI*TST                        ;ENABLE TOGGLE
           +     /Q7*/TST                           ;DISABLE COUNT DURING TEST.
R1        =       Q7                                ;ENABLE Q7 REGISTER OUTPUT
R1.CMBF   =       GND                               ;FOR ROW 1.
                                                    ;
/Q8       :=     /Q8                                ;COUNT EQUATION FOR Q8
          :+:     Q6*Q7*CEI*TST                     ;ENABLE TOGGLE
           +     /Q8*/TST                           ;DISABLE COUNT DURING TEST.
R2        =       Q8                                ;ENABLE Q8 REGISTER OUTPUT
R2.CMBF   =       GND                               ;FOR ROW 2.
                                                    ;
/Q9       :=     /Q9                                ;COUNT EQUATION FOR Q9
          :+:     Q6*Q7*Q8*CEI*TST                  ;ENABLE TOGGLE
           +     /Q9*/TST                           ;DISABLE COUNT DURING TEST.
R3        =       Q9                                ;ENABLE Q9 REGISTER OUTPUT
R3.CMBF   =       GND                               ;FOR ROW 3.
                                                    ;
SIMULATION                                          ;SIMULATION SECTION
TRACE_ON CLK RST MA7 MA8 MA9 MA10                   ;TRACE ESSENTIAL SIGNALS
         MA11 /VSYNC R0 R1 R2 R3                    ;
SETF     /CLK RST CEI /WR                           ;SET INITIAL CONDITIONS
CLOCKF    CLK                                       ;ENABLE COUNT ON
SETF     /RST /TST                                  ;DISABLE ROW R1, R2, R3.
PRLDF    Q0 Q1 Q2 /Q3 Q4                            ;PRELOAD INITIAL STATE
         /Q5 /Q6 /Q7 /Q8 /Q9                        ;TWO COUNTS BEFORE FRAME
FOR      I := 1 TO 40 DO                            ;SYNC. CLOCK INPUT TO VIEW
BEGIN    CLOCKF  CLK                                ;WAVEFORMS OF ROW 0, VSYNC
END                                                 ;AND MEMORY ADDRESSES MA7 TO
                                                    ;MA11.
SETF     TST                                        ;DISABLE TEST MODE, THE ROW
PRLDF    Q0 Q1 Q2 /Q3 Q4                            ;COUNTERS ARE ENABLED.
         /Q5 /Q6 /Q7 Q8 Q9                          ;PROVIDE CLOCK INPUTS TO
FOR      I := 1 TO 40 DO                            ;GENERATE TEST VECTORS TO
BEGIN    CLOCKF  CLK                                ;VERIFY THE NORMAL COUNT
END                                                 ;OPERATION OF THE ROW
SETF     /CEI                                       ;ADDRESS COUNTER.
FOR      I := 1 TO 8 DO                             ;DISABLE COUNT MODE AND
BEGIN    CLOCKF  CLK                                ;APPLY CLOCK INPUT TO
END                                                 ;VERIFY THE COUNTER
TRACE_OFF                                           ;HOLD OPERATION.
```

Figure 9. Design File of Frame Sync Generator (Cont'd.)

Video Shift Register

The third component of this video system (Figures 10 & 11), provides a means of converting parallel data into a high speed serial data stream. The PAL32VX10 implements one such video shift register, where it receives parallel data from a character memory and clocks the serial data out at a rate determined by the pixel clock (DOTCLK). The conversion of parallel character data to a serial stream is synchronized to the character period of the system. The character period is generated by dividing the DOTCLK by eight and this circuit has been conveniently implemented in the same device. The design also offers blanking controls and a reverse video option.

The PAL32VX10 contains ten registers, three of which are used as a binary counter which synchronously counts through eight states at a rate determined by the DOTCLK of the video system. This count to eight determines a character period reference since there are eight pixels (dots) in every character. The outputs S0–S2 are used by the rest of the system as a character period reference.

The three-bit output S2, S1 and S0 of the character period counter are also used internally as a timing reference for loading new character data. When all three S2, S1 and S0 are high, the remaining seven registers of the device Q1–Q7 are loaded with the seven bit character data from inputs D0–D6. The registers used as a counter are Q8, Q9 and Q10. Q10, the least significant bit, is assigned to the output pin S0, Q9 is assigned to pin S1; and Q8 is assigned to pin S2.

The "new character" load command is given as a string statement in the design specification: STRING LOAD 'Q8*Q9*Q10'.

This parallel load is followed by seven shift states when the data is transferred from one register to another. The binary sequence of 0–6 at counter S2–S0 enable this synchronous data shift through registers Q1–Q7, Q7 being the last register in the chain. The VIDEO output of the PAL device is therefore assigned to the last register Q7.

The shift command is developed from the DeMorgan expansion of NOT(LOAD): $\overline{Q8} + \overline{Q9} + \overline{Q10}$.

Additional Control Features

The video shift register has a number of inputs which control the serial output of the video data stream. The BLANK input, when asserted, inhibits the video, causing it to go LOW independent of activity at the parallel data inputs. It is necessary to use the BLANK signal to override active video during line flyback or retrace. The $\overline{VSYNC}$ input also inhibits the loading of video data and suppresses video drive during frame flyback.

The reverse video input REV can be used to create an inverse video display. For normal video this input is LOW. The characters displayed on the video screen will be bright on a dark background. When the reverse video control is set HIGH, the video data is inverted when it is loaded. The display background is then bright, and the displayed characters are dark. The BLANK and VSYNC controls have the same effect of driving the video output LOW during line and frame flyback whether or not the video is reversed.

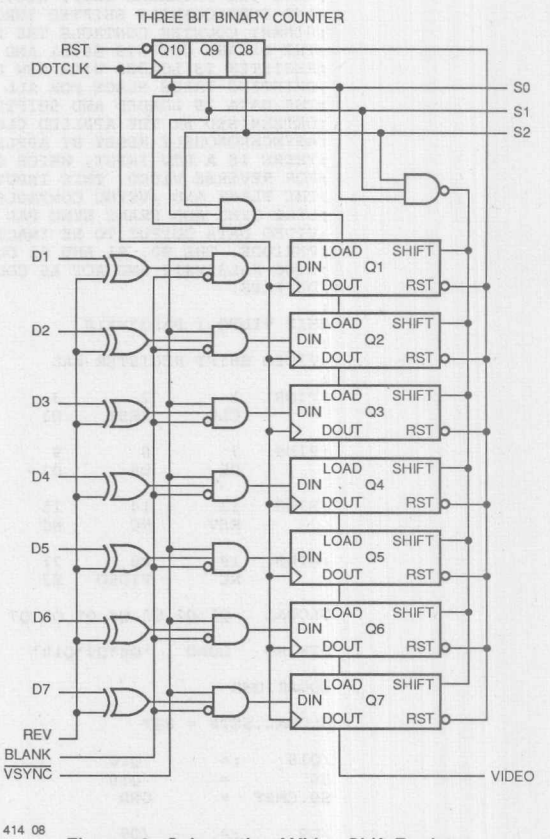


Figure 10. Schematic of Video Shift Register

```
TITLE           VIDEO SHIFT REGISTER.
PATTERN         01
REVISION        02.
AUTHOR          CHRIS JAY.
COMPANY         MMI SANTA CLARA, CA.
DATE            20TH AUGUST 1986.
;
;THE PAL32VX10 HAS BEEN DESIGNED AS A VIDEO SHIFT REGISTER
;TO SUPPORT A THREE PAL SOLUTION TO A SMALL VIDEO CONTROLLER
;CIRCUIT. SEVEN INPUTS FROM THE CHARACTER GENERATOR ARE LOADED
;INTO THE INTERNAL SHIFT REGISTER  THROUGH INPUTS D1 - D7,
;AND SUBSEQUENTLY SHIFTED THROUGH IT. AN INTERNAL THREE BIT
;BINARY COUNTER CONTROLS THE LOAD AND SHIFT ACTIVITY.  WHEN
;THE BINARY OUTPUTS S0,S1 AND S2 ARE ALL HIGH THE SHIFT
;REGISTER IS LOADED WITH NEW DATA FROM THE CHARACTER MEMORY,
;SHIFTING TAKES PLACE FOR ALL THE OTHER BINARY STATES.
;THE DATA IS LOADED AND SHIFTED SYNCHRONOUSLY, AT A RATE
;DETERMINED BY THE APPLIED CLOCK INPUT.  THE DEVICE MAY BE
;ASYNCHRONOUSLY RESET BY APPLYING A LOGIC LOW TO THE /RST PIN.
;THERE IS A REV INPUT, WHICH CAUSES THE DATA TO BE REVERSED
;FOR REVERSE VIDEO. THIS INPUT IS HELD LOW FOR NORMAL DATA.
;THE BLANK AND /VSYNC CONTROLS ARE FEEDBACK INPUTS FROM THE
;LINE SYNC AND FRAME SYNC PAL DEVICE OUTPUTS. THESE CAUSE THE
;VIDEO DATA OUTPUT TO BE INACTIVE DURING LINE AND FRAME SYNC.
;PERIODS. THE S0, S1 AND S2 OUTPUTS DRIVE THE LINE AND FRAME
;SYNC PAL32VX10 AND ACT AS COUNT ENABLE INPUTS TO THOSE
;DEVICES.
;
CHIP VIDEO_1 PAL32VX10
;
;VIDEO SHIFT REGISTER PAL
;
;PINS   1       2       3       4       5       6
        CLK     /RST    D1      D2      D3      D4

;PINS   7       8       9       10      11      12
        D5      D6      D7      BLANK   /VSYNC  GND

;PINS   13      14      15      16      17      18
        REV     NC      NC      NC      NC      NC

;PINS   19      20      21      22      23      24
        NC      VIDEO   S2      S1      S0      VCC

GLOBAL   Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10

STRING   LOAD   'Q8*Q9*Q10'      ;ENABLE LOADING OF CHARACTER DATA.
                                 ;
EQUATIONS                        ;
                                 ;
GLOBAL.SETF = RST                ;ASYNCHRONOUS RESET.
                                 ;
/Q10    :=      Q10              ;INTERNAL STATE COUNTER
S0       =      Q10              ;LEAST SIGNIFICANT BIT Q10.
S0.CMBF  =      GND              ;ENABLE REGISTER TO PIN
                                 ;S0.
/Q9     :=      /Q9              ;INTERNAL STATE COUNTER
        :+:      Q10             ;Q10 = HIGH = TOGGLE.
S1       =      Q9               ;ENABLE REGISTER OUTPUT
S1.CMBF  =      GND              ;TO PIN S1.
                                 ;
/Q8     :=      /Q8              ;INTERNAL STATE COUNTER
        :+:     Q10*Q9           ;Q10*Q9 = HIGH = TOGGLE.
S2       =      Q8               ;ENABLE REGISTER OUTPUT
S2.CMBF  =      GND              ;TO PIN S2.
                                 ;


/Q1     :=      /D1*LOAD*/REV*/BLANK*/VSYNC     ;LOAD DATA INPUT D1.
         +       D1*LOAD*REV*/BLANK*/VSYNC      ;LOAD DATA INPUT /D1.
         +      /Q8*/REV                        ;SHIFT HIGH INTO Q1.
         +      /Q9*/REV                        ;
         +      /Q10*/REV                       ;
         +       BLANK                          ;BLANK DATA DURING
         +       VSYNC                          ;LINE AND FRAME SYNC
                                                ;INPUTS.
```

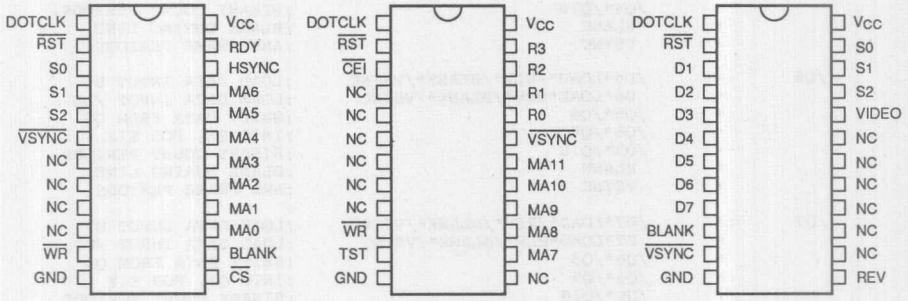
Figure 11. Design File of Video Shift Register

```
/Q2     :=      /D2*LOAD*/REV*/BLANK*/VSYNC     ;LOAD DATA INPUT D2.
        +        D2*LOAD*REV*/BLANK*/VSYNC      ;LOAD DATA INPUT /D2.
        +       /Q1*/Q8                         ;SHIFT DATA FROM Q1
        +       /Q1*/Q9                         ;INTO Q2, FOR SIX
        +       /Q1*/Q10                        ;BINARY COUNT PERIODS.
        +        BLANK                          ;BLANK DURING LINE
        +        VSYNC                          ;AND FRAME PERIODS.
                                                ;
/Q3     :=      /D3*LOAD*/REV*/BLANK*/VSYNC     ;LOAD DATA INPUT D3.
        +        D3*LOAD*REV*/BLANK*/VSYNC      ;LOAD DATA INPUT /D3.
        +       /Q2*/Q8                         ;SHIFT DATA FROM Q2
        +       /Q2*/Q9                         ;INTO Q3, FOR SIX
        +       /Q2*/Q10                        ;BINARY COUNT PERIODS.
        +        BLANK                          ;BLANK DURING LINE
        +        VSYNC                          ;AND FRAME PERIODS.
                                                ;
/Q4     :=      /D4*LOAD*/REV*/BLANK*/VSYNC     ;LOAD DATA INPUT D4.
        +        D4*LOAD*REV*/BLANK*/VSYNC      ;LOAD DATA INPUT /D4.
        +       /Q3*/Q8                         ;SHIFT DATA FROM Q3
        +       /Q3*/Q9                         ;INTO Q4, FOR SIX
        +       /Q3*/Q10                        ;BINARY COUNT PERIODS.
        +        BLANK                          ;BLANK DURING LINE
        +        VSYNC                          ;AND FRAME PERIODS.
                                                ;
/Q5     :=      /D5*LOAD*/REV*/BLANK*/VSYNC     ;LOAD DATA INPUT D5.
        +        D5*LOAD*REV*/BLANK*/VSYNC      ;LOAD DATA INPUT /D5
        +       /Q4*/Q8                         ;SHIFT DATA FROM Q4
        +       /Q4*/Q9                         ;INTO Q5, FOR SIX
        +       /Q4*/Q10                        ;BINARY COUNT PERIODS.
        +        BLANK                          ;BLANK DURING LINE
        +        VSYNC                          ;AND FRAME PERIODS.
                                                ;
/Q6     :=      /D6*LOAD*/REV*/BLANK*/VSYNC     ;LOAD DATA INPUT D6.
        +        D6*LOAD*REV*/BLANK*/VSYNC      ;LOAD DATA INPUT /D6
        +       /Q5*/Q8                         ;SHIFT DATA FROM Q5
        +       /Q5*/Q9                         ;INTO Q6, FOR SIX
        +       /Q5*/Q10                        ;BINARY COUNT PERIODS.
        +        BLANK                          ;BLANK DURING LINE
        +        VSYNC                          ;AND FRAME PERIODS.
                                                ;
/Q7     :=      /D7*LOAD*/REV*/BLANK*/VSYNC     ;LOAD DATA INPUT D7
        +        D7*LOAD*REV*/BLANK*/VSYNC      ;LOAD DATA INPUT /D7
        +       /Q6*/Q8                         ;SHIFT DATA FROM Q6
        +       /Q6*/Q9                         ;INTO Q7, FOR SIX
        +       /Q6*/Q10                        ;BINARY COUNT PERIODS.
        +        BLANK                          ;BLANK DURING LINE
        +        VSYNC                          ;AND FRAME PERIODS.
/VIDEO          = /Q7                           ;ENABLE REGISTER
/VIDEO.CMBF     = GND                           ;OUTPUT Q7 TO VIDEO
                                                ;PIN.
                                                ;
                                                ;
SIMULATION                                      ;START OF SIMULATION
TRACE_ON        CLK S0 S1 S2                    ;TRACE ALL ESSENTIAL
                VIDEO /VSYNC BLANK              ;INPUTS AND OUTPUTS.
                REV Q1 Q2 Q3 Q4 Q5              ;
                Q6 Q7 D1 D2 D3 D4 D5            ;
                D6 D7 RST                       ;SET INITIAL CONDITIONS.
SETF            /CLK RST REV                    ;APPLY ASYNCHRONOUS
                /D1 /D2 D3 D4                   ;RESET. SET REVERSE
                D5 /D6 /D7                      ;VIDEO INACTIVE AND
                BLANK /VSYNC                    ;CREATE ACTIVE BLANK
SETF            /RST                            ;AND REMOVE RESET.
FOR J := 0 TO 5 DO                              ;INPUTS GENERATE FIVE
BEGIN                                           ;LOOPS.
IF J = 1 THEN                                   ;
BEGIN SETF VSYNC /BLANK                         ;SET VSYNC TO TEST
END                                             ;BLANKING.
```

Figure 11. Design File of Video Shift Register (Cont'd.)

```
END                                          ;INPUT. REMOVE ACTIVE
IF J = 3 THEN                                ;VSYNC. CLEAR ALL
BEGIN SETF D1 /D2 D3 /D4 D5                  ;BLANKING CONTROLS.
        /D6 D7 /BLANK /REV                   ;TEST THE LOADING
END                                          ;AND SHIFTING OF
IF J = 4 THEN                                ;DATA. FOR REVERSE
BEGIN SETF REV                               ;VIDEO ACTIVE AND
END                                          ;INACTIVE.
FOR I := 1 TO 8 DO                           ;GENERATE LOOPS
BEGIN                                        ;TO LOAD AND SHIFT
CLOCKF CLK                                   ;DATA.
END                                          ;
END                                          ;
TRACE_OFF                                    ;END OF SIMULATION
```

Figure 11. Design File of Video Shift Register (Cont'd.)



414 09

Figure 12. Pinouts for Various Components of a Small System Graphics Controller

PAL32VX10 Uses Buried Register for Input-Intensive State Machine Designs

AN-160A

The PAL®32VX10 offers design engineers several features not previously found in programmable logic devices (PLDs). Many of these features are found in the output macrocell, which allows over thirty possible configurations. This application note will highlight some of these features, and show how they are used in a Dual Port Video Shift Register.

The PAL32VX10 is a 24-pin PLD with twelve dedicated inputs. The ten I/O macrocells have dual feedback, providing twenty additional array inputs. Each macrocell has the architecture shown in Figure 1.

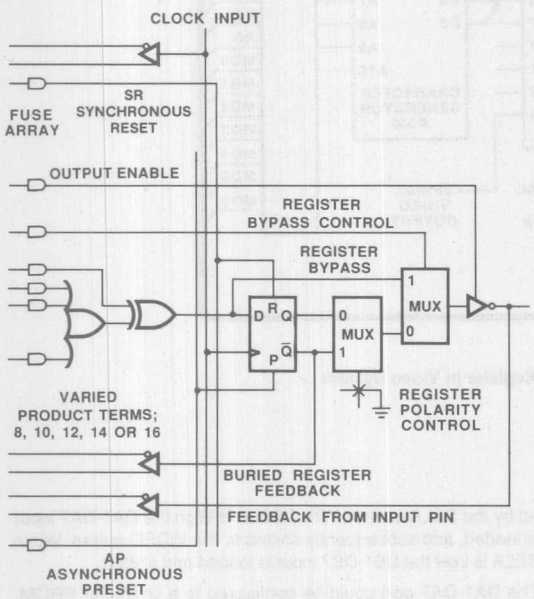


Figure 1. PAL32VX10 Macrocell

Macrocell Description

A D-type flip-flop is clocked by a dedicated input which also feeds the programmable array. The flip-flop data input is fed by a varied number of product terms, from eight to sixteen per output. The sum of these products feeds an Exclusive-OR (XOR) gate controlled by an additional product term. Global reset and preset signals are also controlled by product terms.

A fuse controls flip-flop output polarity. A product term allows the flip-flop to be bypassed, and another product term enables the output. Feedback signals come from the flip-flop and the I/O pin, providing dual feedback.

The device is manufactured with Monolithic Memories' advanced oxide-isolated process, providing high speed and low power consumption. The propagation delay is 30 ns maximum for the standard version and 25 ns maximum for the A-speed version. Power consumption is only 180 mA maximum.

J-K Programmable Function

The XOR gate in each of the ten macrocells provides the designer with the option to extend the functionality of the D-type register to perform J-K and subset functions, such as T-type toggle flip-flops for binary counter applications. This option can aid product term economy considerably. In many state machine designs large numbers of product terms are needed as holding functions for a D-type flip-flop. The hold function is not inherent to the truth table of a D-type register, but in a J-K function it is, so a logic designer can invoke the hold condition without using up any of the product terms in the macrocell.

It can be shown that there exist five logic equations for a J-K function using a D-type flip-flop, an XOR gate, and sum of product inputs;

1. $Q := \overline{Q}*J + Q*\overline{K}$
2. $\overline{Q} := \overline{Q}*\overline{J} + Q*K$
3. $Q := Q :+: (\overline{Q}*J + Q*K)$
4. $\overline{Q} := \overline{Q} :+: (\overline{Q}*J + Q*K)$
5. $Q := \overline{Q} :+: (\overline{Q}*\overline{J} + Q*\overline{K})$

See Appendix 1 for the derivation of these equations.

The key feature to note about these equations is that the first two can be realized from the sum of two product terms but either J or K must be inverted. A DeMorgan expansion of a complex J or K input might result in using too many product terms, and may not fit in the PAL32VX10. This limitation can be circumvented by using equation 3; here, J and K inputs need no inversion but there is a requirement for the XOR operator, as shown in equations 3, 4 and 5. Alternatively, the non-inverted J and K functions may use more terms than the inverted versions, pointing to the use of equations 1, 2, or 5. The macrocell structure is capable of making use of any one of these five equations, allowing the designer to choose the one that makes optimum use of product terms.

The fuse array was extended to 9,738 programmable fuses to accommodate ten additional feedback paths from the ten macrocells. These are the ten buried feedback paths (from the flip-flop) that make it possible to realize buried register state machine designs. Any macrocell whose output pin is programmed as an input still has available its internal feedback path, and the registered cell's functionality need not be lost. The cell could be programmed as a buried register for use in a buried state machine. This feature extends the application of this device to input-intensive designs such as the Dual Port Video Shift Register.

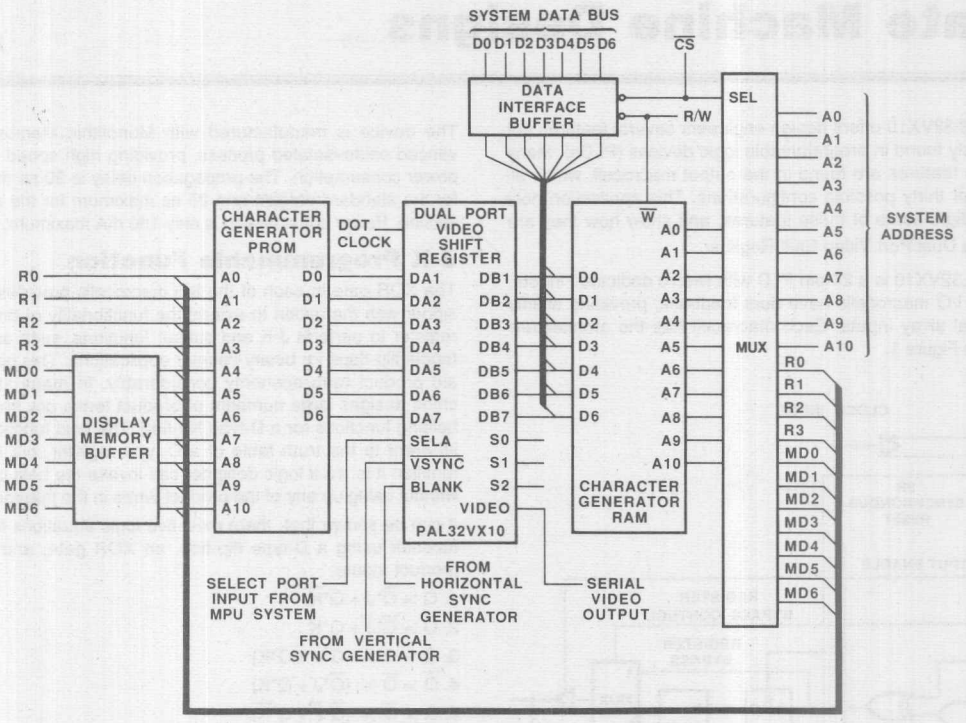


Figure 2. Dual Port Video Shift Register in Video System

Dual Port Video Shift Register

This design highlights the the advantages of using PAL devices for implementing custom designs. This design is shown in the PAL device Design Specification file on page 2-279. Figure 2 shows how the device would be configured to a Microprocessor Unit (MPU) system as a parallel-to-serial converter in video applications.

The PAL32VX10 in the system is capable of supporting one of two parallel input paths from character storage elements. The selected port input is converted to a serial video data stream through the VIDEO output for display on a CRT screen. Events of loading and shifting are synchronized to a Dot Clock input which is applied to pin 1 of the device, and is the rate at which the character dots are displayed on the CRT. There are two input ports, DA1-DA7 and DB1-DB7. One of these inputs is selected by the SELA control. When SELA is High the DA1-DA7 input is loaded, and subsequently shifted to the VIDEO output. When SELA is Low the DB1-DB7 input is loaded and shifted.

The DA1-DA7 port could be configured to a character PROM, the content of which is a fixed character font. A second character memory could be configured to the DB1-DB7 port, which could be a RAM circuit. While the standard character font of the PROM is being displayed on the CRT, the RAM memory could be programmed by the host CPU with a specialized character font. The contents of the RAM could be changed at any time, so a wide variety of custom fonts could be loaded, while the system is displaying the regular standard font stored in the PROM. Switching of the SELA input is controlled by the host system to select the required character memory.

Each of the two input ports is seven bits wide and is designed to interface with character matrix sizes of 7 by 9. However, if the SELA input is controlled by an external register which decodes the state outputs S0, S1 and S2, the two ports may be configured as one single port, fourteen bits wide.

The inputs BLANK and $\overline{\text{VSYNC}}$ are used to force a video blanking output during line and frame flyback. The BLANK input overlaps the line sync signal in a practical application to create front and back porch blanking on the CRT screen.

Of the ten registered macrocells in the PAL32VX10, seven are connected as a parallel load serial shift register to the VIDEO output. The three remaining registers are configured as a three-bit binary counter. The counter outputs S0, S1 and S2 determine whether the device is in load or shift mode. When S0-S2 are all High the shift register loads character information; otherwise, shifting takes place for the seven remaining states of the binary counter. The output pins S0, S1 and S2 are connected to the buried registers Q10, Q9 and Q8 respectively, and are assigned in the PAL device Design Specification by S0 = Q10, etc. Similarly, the VIDEO output pin is assigned register Q7. Registers Q1 to Q6 remain buried; the output pins normally associated with these register cells have been assigned as the dedicated DB port inputs DB2-DB7.

The three-bit binary counter has been designed using a T-type toggle register. Choosing Equation 4 and equating J = K = 1 for the Toggle function and J = K = 0 for the Hold function, in general J = K = T. The new input for a Toggle function becomes T and the equation becomes:

6. $\overline{Q} := \overline{Q} :+: (\overline{Q} + Q)*T$

but since

7. $(\overline{Q} + Q) = 1$

the equation becomes:

8. $\overline{Q} := \overline{Q} :+: T.$

The basic function in a binary counter is that a flip-flop must remain in a Hold state until the product term made up of all of the less significant flip-flops in the count sequence becomes a logic High. So, the equation for Q8 in the design specification becomes:

9. $\overline{Q8} := \overline{Q8} :+: Q9*Q10.$

When Q9*Q10 = 0 flip-flop Q8 holds its contents, and when Q9*Q10 = 1 Q8 changes state.

The PAL device Design Specification shows that eighteen inputs and four outputs are required, but the buried register feature has not wasted the macrocells whose associated pins have been used as inputs. The design has been programmed with two other PAL devices (not shown here), one as a line sync generator and one as a frame sync generator, to form the basis of a low-cost video system.

A simulation section has been included with the design to verify functional operation before the Design Specification is committed to the fuse array of the PAL device. It is recommended that the A-speed part is used in this specific application to accommodate good resolution of the displayed video information.

The oscilloscope photograph in Figure 4 shows the S0, S1 and S2 outputs, running from the top trace down, and the serial VIDEO data on the bottom trace. The Load event takes place when all state outputs are High. For the first section of the trace, port DB7-DB1 is loaded and subsequently shifted through the buried register file. The cursor on the right marks the end of one Load/Shift cycle. The bit pattern loaded into DB7-DB1 is HLLHHHL. In the second half of the the trace, port DA7-DA1 is selected. The bit pattern of LLHLHLH is clocked into the DA7-DA1 port and shifted through to the VIDEO output.

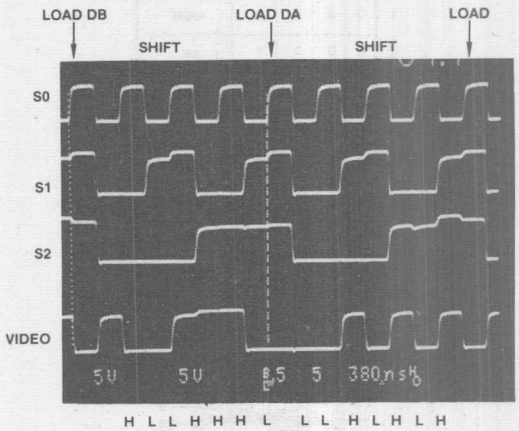
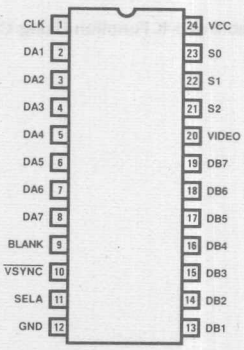


Figure 4. Video Shift Register Waveforms

PAL32VX10

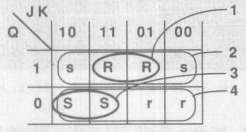


Dual Port Video Shift Register

Derivation of the J-K Function

The complete truth table of a J-K function is shown in Table 1. Inputs J and K are tabulated with the current state of the register (QN) and the required next state (QN+1). Equations 1 and 2 from the text are derived by directly grouping ones (S and s) or zeros (R and r) in the truth table. The other equations for the J-K function are derived by tabulating active 'SET' and 'RESET' functions and passive 'set' and 'reset' functions.

	J	K	Q_N	Q_{N+1}	FUNCTION
1	0	0	0	0	reset
2	0	0	1	1	set
3	1	0	0	1	SET
4	1	0	1	1	set
5	0	1	0	0	reset
6	0	1	1	0	RESET
7	1	1	0	1	SET
8	1	1	1	0	RESET



Q = /Q*J + Q*/K EQ. 1
/Q = /Q*/J + Q*K EQ. 2
Q = Q :+: (/Q*J + Q*K) EQ. 3
/Q = /Q :+: (/Q*J + Q*K) EQ. 4
Q = /Q :+: (/Q*/J + Q*/K) EQ. 5

Table1. Truth Table of J-K Function Using Change of State.

Identifying Transition and Hold conditions as well as logic High and Low states enables the designer to draw a Karnaugh map for an XOR-gate register input. The XOR gate has the capability of a Hold function for the remain in 'set' and 'reset' states, and a Change-of-State function for active 'SET' and 'RESET'. To derive Equation 3,

Q := Q :+: ($\overline{Q}$*J + Q*K),

the following equation is considered:

Q := s :+: (R + S),

or, "Q holds unless an active transition is true." The entries in the equation show Hold and Transition states. When both 'S' and 'R' are inactive Low the register Q is in a Hold condition by the 's' entry in the equation. If 's' and 'R' become High simultaneously the result will be an active Reset after the following clock edge; for this reason minimization of the 's' equation can include any 'R' entries in the Karnaugh Map. Also, the states of 'S' and 's' are mutually exclusive, so with 's' inactive and 'S' active the output will transition to a Set condition.

Minimize all the Hold 'set' states but include the active 'RESET' states for the entry to the left of the XOR function.This gives Q as shown in group 2 of the Karnaugh map. Then identify the change-of-state entries shown by the capital letters 'S' and 'R'. Minimization achieved in groups 1 and 3 are shown to the right of the equation.

With regard to the equation for Q a similar procedure applies:

Q := r :+: (S + R)

This time all the passive 'reset' conditions are minimized with active 'SET' states, and XORed with minimization of active 'SET' and 'RESET' states (group 4). This leads to Equation 4.

Equation 5 is derived by examining the equation for "Q will toggle unless a Hold is true", or

Q := $\overline{Q}$:+: (s + r);

the derivation is similar to those above.

```
TITLE           DUAL PORT VIDEO SHIFT REGISTER.
PATTERN         01B
REVISION        01
AUTHOR          CHRIS JAY.
COMPANY         MMI SANTA CLARA, CA.
DATE            15TH MARCH 1987.
;
;THE PAL32VX10 HAS BEEN DESIGNED AS A VIDEO SHIFT REGISTER
;WITH A DUAL SEVEN BIT PORT INPUT.  PARALLEL CHARACTER DATA
;MAY BE LOADED FROM A FIXED CHARACTER ROM VIA PORT INPUTS
;DA1 - DA7, WHILE THE SELA INPUT IS HIGH.  PORT INPUTS DB1-DB7
;MAY BE CONNECTED TO A CHARACTER RAM, THE CONTENTS OF WHICH
;CAN BE CHANGED BY THE HOST MICROPROCESSOR.  THIS ENABLES THE
;VIDEO SYSTEM DISPLAY A STANDARD CHARACTER FONT WHILE DYNAMICALLY
;CHANGING CUSTOM CHARACTER FONTS IN THE CHARACTER RAM.  WHEN A
;NEW CHARACTER DATA IS LOADED INTO THE RAM THE SELA INPUT CAN BE
;DRIVEN LOW TO SELECT THE DB1-DB7 PORT INPUT.  THE ON CHIP
;STATE COUNTER S0, S1 AND S2 FORMS A SYNCHRONOUS BINARY COUNT.
;STATES 0 TO 6 REPRESENT SHIFT ACTIVITY IN THE SEVEN BIT VIDEO
;SHIFT REGISTER.  WHEN THE COUNTER REACHES A COUNT OF SEVEN THE
;NEXT CHARACTER DATA IS LOADED INTO THE REGISTER VIA PORT DA OR DB.
;THERE ARE TWO BLANKING INPUTS /VSYNC AND BLANK.  WHEN /VSYNC IS
;LOW THE CONTENTS OF THE VIDEO SHIFT REGISTER ARE LOADED WITH LOGIC
;ZEROS.  THIS IS ALSO THE CASE WHEN BLANK IS ACTIVE HIGH.  THE
;VSYNC AND BLANK INPUTS SHOULD BE USED DURING FRAME AND HORIZONTAL
;FLYBACK, RESPECTIVELY.
;
CHIP VIDEO_2 PAL32VX10
;
;DUAL PORT VIDEO SHIFT REGISTER PAL
;
;PINS    1       2       3       4       5       6
         CLK     DA1     DA2     DA3     DA4     DA5

;PINS    7       8       9       10      11      12
         DA6     DA7     BLANK   /VSYNC  SELA    GND

;PINS    13      14      15      16      17      18
         DB1     DB2     DB3     DB4     DB5     DB6

;PINS    19      20      21      22      23      24
         DB7     VIDEO   S2      S1      S0      VCC

GLOBAL    Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10

STRING    LOAD    'Q8*Q9*Q10'       ;ENABLE LOADING OF CHARACTER DATA.
                                    ;
EQUATIONS                           ;
                                    ;
/Q10      :=       Q10              ;INTERNAL STATE COUNTER
S0         =       Q10              ;LEAST SIGNIFICANT BIT
S0.CMBF    =      GND               ;Q10. ENABLE REGISTER
                                    ;OUTPUT FOR PIN S0.
/Q9       :=      /Q9               ;INTERNAL STATE COUNTER
          :+:      Q10              ;Q10 = HIGH = TOGGLE.
S1         =       Q9               ;ENABLE PIN S1
S1.CMBF    =      GND               ;REGISTERED OUTPUT.
```

2

```
                                        ;
/Q8      :=     /Q8                     ;INTERNAL STATE COUNTER
         :+:    Q10*Q9                  ;Q10*Q9 = HIGH = TOGGLE.
S2        =     Q8                      ;ENABLE PIN S2 AS A
S2.CMBF   =     GND                     ;REGISTERED OUTPUT.

/Q1      :=     /DA1*LOAD*SELA*/BLANK*/VSYNC    ;LOAD DATA INPUT DA1.
          +     /DB1*LOAD*/SELA*/BLANK*/VSYNC   ;LOAD DATA INPUT DB1.
          +     /Q8                             ;SHIFT HIGH INTO Q1.
          +     /Q9                             ;WHEN Q10,Q9 AND Q8
          +     /Q10                            ;ARE NOT ALL HIGH
          +      BLANK                          ;SIMULTANEOUSLY.
          +      VSYNC                          ;LINE/FRAME BLANKING
                                                ;
/Q2      :=     /DA2*LOAD*SELA*/BLANK*/VSYNC    ;LOAD DATA INPUT DA2.
          +     /DB2*LOAD*/SELA*/BLANK*/VSYNC   ;LOAD DATA INPUT DB2.
          +     /Q1*/Q8                         ;SHIFT DATA FROM Q1
          +     /Q1*/Q9                         ;INTO Q2.
          +     /Q1*/Q10                        ;
          +      BLANK                          ;LINE AND FRAME
          +      VSYNC                          ;BLANKING.
                                                ;
/Q3      :=     /DA3*LOAD*SELA*/BLANK*/VSYNC    ;LOAD DATA INPUT DA3.
          +     /DB3*LOAD*/SELA*/BLANK*/VSYNC   ;LOAD DATA INPUT DB3
          +     /Q2*/Q8                         ;SHIFT DATA FROM Q2
          +     /Q2*/Q9                         ;INTO Q3.
          +     /Q2*/Q10                        ;
          +      BLANK                          ;LINE AND FRAME
          +      VSYNC                          ;BLANKING.
                                                ;
/Q4      :=     /DA4*LOAD*SELA*/BLANK*/VSYNC    ;LOAD DATA INPUT DA4.
          +     /DB4*LOAD*/SELA*/BLANK*/VSYNC   ;LOAD DATA INPUT DB4
          +     /Q3*/Q8                         ;SHIFT DATA FROM Q3
          +     /Q3*/Q9                         ;INTO Q4.
          +     /Q3*/Q10                        ;
          +      BLANK                          ;LINE AND FRAME
          +      VSYNC                          ;BLANKING.
                                                ;
/Q5      :=     /DA5*LOAD*SELA*/BLANK*/VSYNC    ;LOAD DATA INPUT DA5.
          +     /DB5*LOAD*/SELA*/BLANK*/VSYNC   ;LOAD DATA INPUT DB5
          +     /Q4*/Q8                         ;SHIFT DATA FROM Q4
          +     /Q4*/Q9                         ;INTO Q5.
          +     /Q4*/Q10                        ;
          +      BLANK                          ;LINE AND FRAME
          +      VSYNC                          ;BLANKING.
                                                ;
/Q6      :=     /DA6*LOAD*SELA*/BLANK*/VSYNC    ;LOAD DATA INPUT DA6.
          +     /DB6*LOAD*/SELA*/BLANK*/VSYNC   ;LOAD DATA INPUT DB6.
          +     /Q5*/Q8                         ;SHIFT DATA FROM Q5
          +     /Q5*/Q9                         ;INTO Q6.
          +     /Q5*/Q10                        ;
          +      BLANK                          ;LINE AND FRAME
          +      VSYNC                          ;BLANKING.
                                                ;
```

```
/Q7      :=      /DA7*LOAD*SELA*/BLANK*/VSYNC     ;LOAD DATA INPUT DA6.
          +      /DB7*LOAD*/SELA*/BLANK*/VSYNC    ;LOAD DATA INPUT DB6.
          +      /Q6*/Q8                          ;SHIFT DATA FROM Q6
          +      /Q6*/Q9                          ;INTO Q7.
          +      /Q6*/Q10                         ;
          +       BLANK                           ;LINE AND FRAME
          +       VSYNC                           ;BLANKING.
/VIDEO           = /Q7                            ;
/VIDEO.CMBF      = GND                            ;OUTPUT Q7 TO VIDEO.


SIMULATION                                        ;START OF SIMULATION
TRACE_ON         CLK S0 S1 S2                     ;TRACE ALL ESSENTIAL
                 VIDEO /VSYNC BLANK SELA          ;
                 Q1 Q2 Q3 Q4 Q5 Q6                ;INPUTS AND OUTPUTS.
                 DA1 DA2 DA3 DA4 DA5 DA6 DA7      ;
                 DB1 DB2 DB3 DB4 DB5 DB6 DB7      ;
PRLDF            Q1 Q2 Q3 Q4 Q5  Q6 Q7 Q8 Q9 Q10  ;PRELOAD REGISTERS
SETF             /CLK                             ;SET INITIAL CONDITIONS.
                 DA1 /DA2 DA3 /DA4 DA5 /DA6 DA7   ;ON DATA A INPUT PORT
                 /DB1 DB2 /DB3 DB4 /DB5 DB6 /DB7  ;AND DATA B INPUT PORT.
                 SELA BLANK /VSYNC                ;ENABLE A PORT AND
FOR I := 0 TO 7 DO                                ;BLANK INPUT.
BEGIN CLOCKF CLK                                  ;BEGIN THE DOT CLOCK
END                                               ;INPUT FOR EIGHT CLOCK
SETF    /SELA                                     ;CYCLES. SELECT B INPUT
FOR I := 0 TO 7 DO                                ;ENABLE CLOCK FOR EIGHT
BEGIN CLOCKF CLK                                  ;CLOCK CYCLES.
END                                               ;
SETF    /BLANK SELA                               ;REMOVE BLANKING INPUT
FOR I := 0 TO 7 DO                                ;SELECT A PORT CLOCK
BEGIN CLOCKF CLK                                  ;FOR EIGHT CYCLES.
END                                               ;
SETF    /SELA                                     ;ENABLE B INPUT PORT
FOR I := 0 TO 7 DO                                ;CLOCK FOR EIGHT CLOCK
BEGIN CLOCKF CLK                                  ;CYCLES.
END                                               ;
SETF    VSYNC                                     ;ENABLE VSYNC INPUT
FOR I := 0 TO 7 DO                                ;CLOCK FOR EIGHT
BEGIN CLOCKF CLK                                  ;CLOCK CYCLES.
END                                               ;
TRACE_OFF                                         ;END OF SIMULATION
```

IMPORTANT.
NOTES

Digital Signal Processing

The field of Digital Signal Processing (DSP) was originally limited to the digital filtering and processing of analog signals. Today, the field encompasses a wide range of numeric processing applications, primarily due to the versatility of the Fast Fourier Transform algorithm with its accompanying repetitive addition and multiplication operations.

The ability to perform multiplication and addition in a single clock cycle is the key to achieving high-speed mathematics in DSP hardware. When executing math-intensive algorithms, even the slowest DSP components are at least ten times faster than conventional microprocessors. When a DSP solution is used, it is usually for speed considerations. Conventional microprocessors are not optimized for arithmetic, and have to simulate math through software. This simulation tends to require several clock cycles. DSP systems use specialized hardware, including a multiplier and adder, that allow them to execute numeric operations in a single cycle. A typical DSP system is shown in Figure 1.

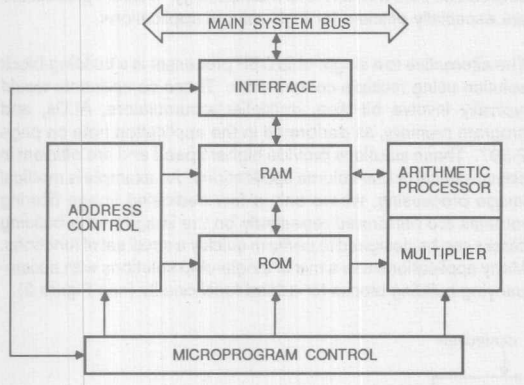


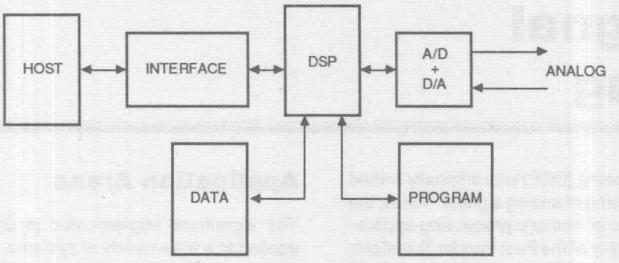
Figure 1. High-Performance DSP System

In addition, the architecture must try to provide data to the multiplier and adder at every clock cycle, for the highest efficiency. This can be done by using an architecture where data and instructions are stored separately and have their own buses, one for data and one for instructions. If two data buses are used, two data elements can be moved to the multiplier or adder in parallel for execution, while the CPU is fetching an instruction. Another efficient architecture uses pipelining, allowing the processor to work on subsections of the operation in parallel.

Application Areas

The algorithms implemented in DSP components are being applied to a wide variety of systems. Examples include standard signal processing for applications from compact disc players to artificial intelligence, matrix-manipulation algorithms for graphics systems, filtering algorithms used in image-processing systems, and floating-point algorithms used in workstations and scientific computers. Many of these applications are often categorized under "numerical processing" or "number crunching" rather than DSP, but any algorithm that makes extensive use of repetitive multiplication and addition operations can benefit from DSP hardware. The primary application area is communications, with military applications also a major part of the market. The application areas are listed below.

- Communications
  - –Codecs
  - –Modems
  - –Multiplexers/Demultiplexers
  - –Echo Cancellation
  - –Filtering
  - –Gain equalization
- Military
  - –Radar/Sonar
  - –Guidance and Control
  - –Telemetry
  - –Electronic Warfare/Countermeasures
- Computers
  - –PC-Board Add-Ons for PCs
  - –Minicomputers/Workstations
  - –Parallel/Array Processors
- Graphics and Image Processing
  - –CAD/CAE
  - –Image Enhancement and Restoration
  - –Image Subtraction, Compression, Decompression
  - –CAT Scanning
- Industrial Automation
  - –CAM
  - –Vision Systems/Robotic Vision and Other Senses
  - –Artificial Intelligence
- Voice Data Entry/Synthesis
- Test Equipment
- Process Control
- Consumer Products
  - –Television
  - –Compact Disc Players



411 02

Figure 2. Typical General-Purpose DSP Processor System

Speed versus Other Requirements

As mentioned, most DSP designs are chosen because of speed. However, speed is not always a critical element once a DSP solution is selected: the slowest DSP design will still be faster than the fastest conventional microprocessor design.

Speed remains a critical element in most communications applications. An exception is modems, which require single-chip solutions for size reduction and do not require high speed to handle voice frequencies. Speed is also critical in computer applications, but price and performance advantages are also important. DSP add-in boards require small size, and therefore single-chip solutions. In military applications, however, price is less critical, and designs are typically customized for performance.

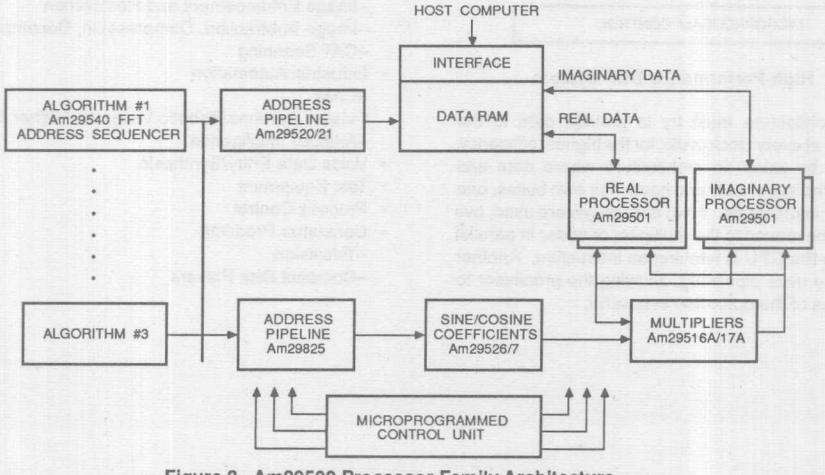
Consumer applications use custom single-chip solutions for both size reduction and cost reduction; these are usually produced by a captive supplier. Many of the other applications require a flexible solution and a short development time, both efficiently provided by programmable logic devices.

Single Chip versus Building Block

Similar to the beginnings of the conventional microprocessor, dedicated DSP processors are being introduced along with development software. Single-chip DSP solutions offer easier software development, smaller size, and lower cost. Several DSP processors are commercially available, and are taking the place of conventional microprocessors in DSP systems (Figure 2). Most are general-purpose processors, but application-specific processors are also being developed.

In addition to these dedicated DSP processors, custom processors can be built with an ASIC methodology. Custom processors are especially efficient for high-volume applications.

The alternative to a single-chip DSP processor is a building-block solution using multiple components. These components would typically involve bit-slice, multiplier/accumulators, ALUs, and program memory, as performed in the application note on page 2-307. These solutions provide higher speed and are efficient in speed-critical or low-volume applications. An example is medical image processing, where only a few dedicated image filtering routines are performed repeatedly on the images—the building blocks can be designed to perform quickly a small set of functions. Many applications use a mix of single-chip solutions with accompanying building blocks for added functionality (see Figure 3).



411 03

Figure 3. Am29500 Processor Family Architecture.

PLD Applications

PLDs are useful both in single-chip and building block designs. If DSP components are used as accelerators in conjunction with standard microprocessors, the interface logic can be placed into PLDs.

When supporting DSP processors, PLDs perform functions similar to those supporting a general purpose microprocessor. These include address decoding, interrupt control, DMA control, and bus control. See the section on Microprocessor-Based Systems for more detail.

Programmable logic devices are also useful in the building-block applications. PLDs often perform functions similar to their use in standard bit-slice designs, as counters or sequencers.

The need for a continuous supply of data to the numeric components of the DSP system provides opportunities for PLDs in both building-block and single-chip designs. Handshaking between components, especially between the numeric chips and the memory or sequencer, is also required, and with multiple buses in the system, complex bus interface logic is needed.

The application note on page 2-307 shows PAL devices providing several functions for a building-block DSP system which implements an audio spectrum analyzer. The PAL devices create the sequencer for program execution, multiplex the data outputs to an oscilloscope, and provide several miscellaneous control functions.

PLDs provide an effective solution in any type of DSP architecture. General-purpose DSP processors need PLDs to perform typical processor support functions, while ASIC processors may need additional logic to provide the proper interface to other chips. In building block systems, PLDs perform both the basic control functions and interface functions. With limited availability of software development tools for most DSP applications, programmable logic devices offer the flexibility to implement last-minute design changes to adapt to the needs of the software and application.

Waveform Generator

A simple waveform generator can be built with a PAL device addressing locations within a PROM. Using a PAL20X10 device to address locations in a registered PROM (Monolithic Memories 63RA1681), digital data may be accessed and fed to a Digital to Analog Converter (DAC). The locations in the memory are programmed with appropriate data to provide a sinusoidal, triangular, sawtooth or step function output. A square wave output may be generated directly from the PAL device itself.

Digital vs. Analog Waveform Generation

There are advantages and disadvantages in generating analog functions by this method when considered against a dedicated analog circuit design. The main advantages favoring the digital method of waveform generation involve controllability. In this design example phase shift generation can be controlled within a tolerance of one degree. The PAL device has been programmed as a 180 degree up/down counter. An initial count and up/down control may be loaded into the counter so precise phase shifts can be generated. When considering the phase shift of a sinusoidal waveform in an analog circuit, complex Resistor-Capacitor (RC) networks might need to be added to generate phase shifting.

Another advantage is the ability of the system to generate a wide range of waveform types, and the convenience of generating new waveforms by simply re-programming a PROM, or programming blank locations in the existing PROM. Also, if it were a requirement to generate a burst of any number of cycles, this could be controlled by a digital interface from a microprocessor.

The disadvantages associated with the digital circuit would be the generation of quantization noise due to the encoding of a digital code into a close approximation of the analog conversion. For example, an eight-bit digital code can be converted to 256 different voltage levels. Supposing a 10 volt signal were to be evenly encoded into 256 discrete levels, the difference between each level would be about 39 mV. The approximation of the digital signal to its analog equivalent gives rise to quantization noise, in this example 39 mV.

A way of reducing the quantization noise is to generate a larger digital code. For a 10-volt dynamic range encoded by a fourteen-bit digital code, the quantization noise is reduced to 0.61 mV. Of course there are problems in increasing the resolution of the digital code in the respect that a much higher sampling frequency would be required to generate the analog output of a given period.

Additional to the problem of quantization noise is the noise generated by the super-imposition of elements of the master sampling clock on the analog wave output. Fortunately the frequency of this clock is much higher than the analog output, for a resolution of eight bits, and may be filtered out with a simple RC filter.

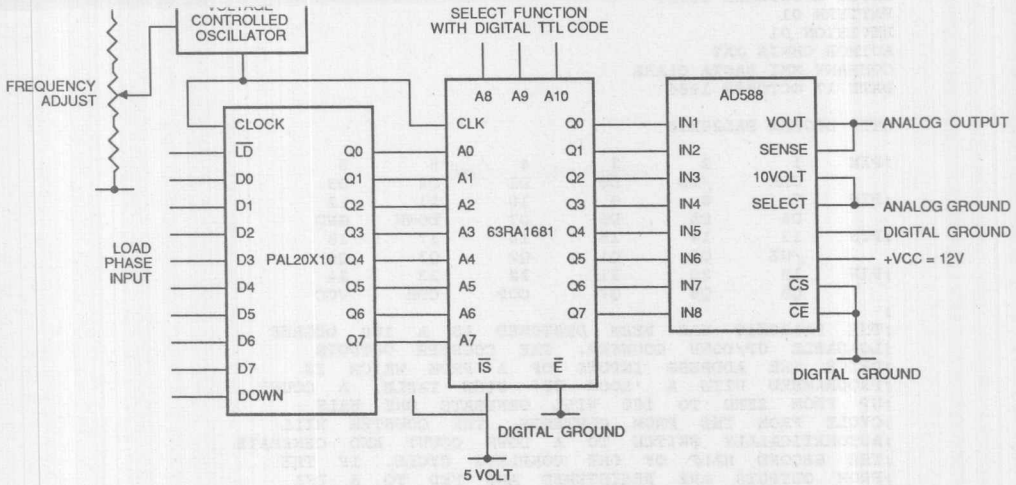
In summary, the advantages in a digital waveform generator are in the controllability of phase shifting, and the number of cycles generated. Also, the ability to guard against drift might be an advantage. With an analog system, purity of signal output and the ability to generate higher frequencies are the main advantages.

PAL Device Implementation

The diagram in figure 1 shows the circuit used to create the waveforms. Photographs of the waveforms are provided in figure 2. The different functions generated were selectable by a unique digital code applied to the higher order address inputs A8, A9, and A10 of the PROM. The lower order addresses were driven from the PAL20X10; outputs Q0–Q7 were fed to address inputs A0–A7. The design specification of the 180 Degree Up/Down counter (page 2-288) was encoded into the PAL device.

Look-Up Table Generation

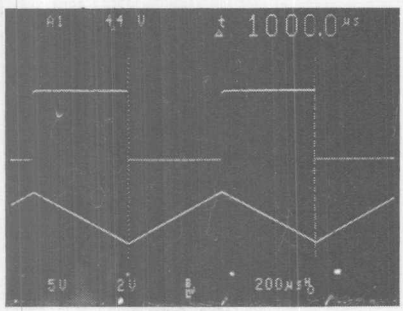
To generate the hexadecimal look-up table, which was programmed into the PROM, a small TURBO PASCAL program was written to provide a conversion from the angle (degrees) to the sine of the angle, in a hexadecimal value (page 2-290). The program generated the nearest hexadecimal value to 100 (sine(x) +1), for a range of 180 degrees, at a resolution of one degree. The sawtooth and triangular waveforms were simple identity functions; the PROM location address was programmed into its own contents. The step waveform was programmed as constant values in a number of successive adjacent locations.



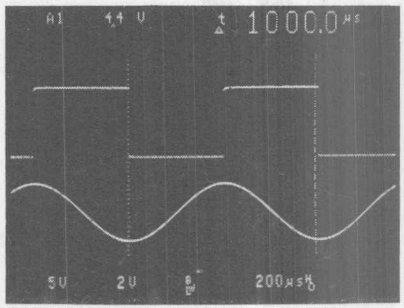
439 01

2

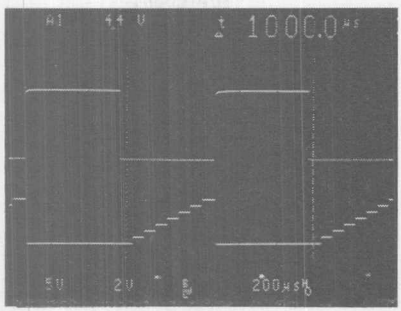
Figure 1.



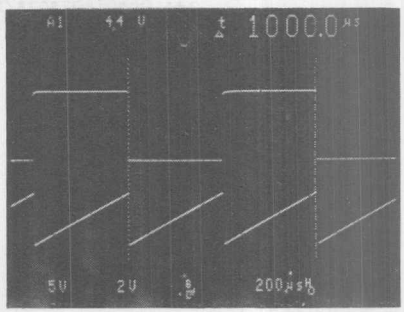
Triangular Waveform



Sinewave



Step Waveform



Sawtooth Waveform

Figure 2.

```
TITLE 180DEGREE COUNT
PATTERN 01
REVISION 01
AUTHOR CHRIS JAY
COMPANY MMI SANTA CLARA
DATE 17 OCTOBER 1986

CHIP DEGREE PAL20X10

;PIN    1       2       3       4       5       6
        CLK     /LD     D0      D1      D2      D3
;PIN    7       8       9       10      11      12
        D4      D5      D6      D7      DOWN    GND
;PIN    13      14      15      16      17      18
        /OE     Q0      Q1      Q2      Q3      Q4
;PIN    19      20      21      22      23      24
        Q5      Q6      Q7      CUP     CDN     VCC
;
;THE  PAL20X10  HAS  BEEN  DESIGNED  AS  A  180  DEGREE
;LOADABLE  UP/DOWN  COUNTER.  THE  COUNTER  OUTPUTS
;DRIVE  THE  ADDRESS  INPUTS  OF  A  PROM  WHICH  IS
;PROGRAMMED  WITH  A  'LOOK  UP'  SINE  TABLE.  A  COUNT
;UP  FROM  ZERO  TO  180  WILL  GENERATE  ONE  HALF
;CYCLE  FROM  THE  PROM  CONTENTS.  THE  COUNTER  WILL
;AUTOMATICALLLY  SWITCH  TO  A  DOWN  COUNT  AND  GENERATE
;THE  SECOND  HALF  OF  ONE  COMPLETE  CYCLE.  IF  THE
;PROM  OUTPUTS  ARE  REGISTERED  AND  FED  TO  A  D/A
;CONVERTER  A  CONTINOUS  SINUSOIDAL  SIGNAL  WILL  BE
;GENERATED.  COUNTING  CAN  BE  INTERRUPTED  WITH  A
;LOAD.  PHASE  SHIFTING  OF  THE  SINUSIODAL  WAVEFORM
;MAY  BE  ACHIEVED  WITH  A  RESOLUTION  OF  1  DEGREE.
;
EQUATION

/Q0     := Q0*/LD                               ;COUNT
        +  /D0*LD                               ;LOAD
                                                ;
/Q1     := /Q1*/LD                              ;HOLD
        +  /D1*LD                               ;LOAD
        :+: Q0*/LD*CUP                          ;UP COUNT
        +  /Q0*/LD*/CUP                         ;DOWN COUNT
                                                ;
/Q2     := /Q2*/LD                              ;HOLD
        +  /D2*LD                               ;LOAD
        :+: Q0*Q1*/LD*CUP                       ;UP COUNT
        +  /Q0*/Q1*/LD*/CUP                     ;DOWN COUNT
                                                ;
/Q3     := /Q3*/LD                              ;HOLD
        +  /D3*LD                               ;LOAD
        :+: Q0*Q1*Q2*/LD*CUP                    ;UP COUNT
        +  /Q0*/Q1*/Q2*/LD*/CUP                 ;DOWN COUNT
                                                ;
/Q4     := /Q4*/LD                              ;HOLD
        +  /D4*LD                               ;LOAD
        :+: Q0*Q1*Q2*Q3*/LD*CUP                 ;UP COUNT
        +  /Q0*/Q1*/Q2*/Q3*/LD*/CUP             ;DOWN COUNT
                                                ;
```

```
/Q5     := /Q5*/LD                                          ;HOLD
        +  /D5*LD                                           ;LOAD
        :+: Q0*Q1*Q2*Q3*Q4*/LD*CUP                          ;UP COUNT
        +  /Q0*/Q1*/Q2*/Q3*/Q4*/LD*/CUP                     ;DOWN COUNT
                                                            ;
/Q6     := /Q6*/LD                                          ;HOLD
        +  /D6*LD                                           ;LOAD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*/LD*CUP                       ;UP COUNT
        +  /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/LD*/CUP                 ;DOWN COUNT
                                                            ;
/Q7     := /Q7*/LD                                          ;HOLD
        +  /D7*LD                                           ;LOAD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*/LD*CUP                    ;UP COUNT
        +  /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/LD*/CUP             ;DOWN COUNT
                                                            ;
/CUP    := /CUP*/LD                                         ;COUNT UP
        +  DOWN*LD                                          ;REGISTER
        :+: /Q7*/Q6*/Q5*/Q4*/Q3*/Q2*/Q1*Q0*/LD*/CUP         ;
        +   Q7*/Q6*Q5*Q4*/Q3*/Q2*Q1*Q0*/LD*CUP              ;
                                                            ;
/CDN    := CUP*/LD                                          ;COUNT DOWN
        +  /DOWN*LD                                         ;REGISTER
        :+: /Q7*/Q6*/Q5*/Q4*/Q3*/Q2*/Q1*Q0*/LD*/CUP         ;
        +   Q7*/Q6*Q5*Q4*/Q3*/Q2*Q1*Q0*/LD*CUP              ;
                                                            ;
SIMULATION                                                  ;
TRACE_ON CLK /DOWN LD CUP CDN                               ;
         Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7                            ;
SETF    /CLK LD OE D0 D1 D2                                 ;LOAD 03H
        /D3 /D4 /D5 /D6 /D7 DOWN                            ;AND DOWN
CLOCKF                                                      ;COUNT. SET
SETF    /LD                                                 ;INTO COUNT
CLOCKF                                                      ;MODE AND
FOR I := 0 TO 15 DO                                         ;GENERATE
BEGIN CLOCKF                                                ;VECTORS TO
END                                                         ;TEST DOWN
SETF    /CLK LD /D0 /D1 /D2                                 ;TO UP TRANS-
       /D3 D4 D5 /D6 D7 /DOWN                               ;-ITION. LOAD
CLOCKF                                                      ;B0H AND UP
SETF    /LD                                                 ;COUNT.
CLOCKF                                                      ;GENERATE
FOR I := 0 TO 15 DO                                         ;VECTORS TO
BEGIN CLOCKF                                                ;TEST UP TO
END                                                         ;DOWN TRANS-
TRACE_OFF                                                   ; -ITION.
```

```
(*********************************************************************)
(*                                                                   *)
(* The program 'sinetable' has been designed to generate             *)
(* the closest possible HEXIDECIMAL value to 100 X [sine(x) + 1      *)
(* for -90 <= x => +90, in steps of delta x = 1 degree. The          *)
(* HEXIDECIMAL PROM address locations have been evaluated as         *)
(* as n = x + 90. The program will generate a SINE.DAT file          *)
(* which will contain a table of PROM address locations from         *)
(* from 00 to B4 HEX, and the contents of each location as the       *)
(* closest possible sine value as computed in the above equation.    *)
(* Program written by:                                               *)
(* Chris Jay and Ser-Hou Kuang. MMI Santa Clara.                     *)
(*                                                                   *)
(*********************************************************************)

const
   hexsize = 10;
type
   hex_type = packed array  [1..hexsize] of char;

var
SinFile: text;
i,k:integer;
s,x,f:real;
hexar,hexcon : hex_type;

(*********************************************************************)
(*                                                                   *)
(* Procedure to convert decimal values into hexadecimal values       *)
(*                                                                   *)
(*********************************************************************)

Procedure DecToHex (dec : integer; var tohex: hex_type);
var
   i, dig, idx, next : integer;

 Function hex (i : integer): char;
  begin
     case i of
       0 : hex := '0';
       1 : hex := '1';
       2 : hex := '2';
       3 : hex := '3';
       4 : hex := '4';
       5 : hex := '5';
       6 : hex := '6';
       7 : hex := '7';
       8 : hex := '8';
       9 : hex := '9';
      10 : hex := 'A';
      11 : hex := 'B';
      12 : hex := 'C';
      13 : hex := 'D';
      14 : hex := 'E';
      15 : hex := 'F';

     end;
    end;
```

```
begin
   for i := 1 to hexsize do
      tohex [i] := ' ';
   idx  := hexsize;
   dig  := dec;
   next := dig div 16;
   while (next <> 0) and  (idx > 0) do
      begin
        ToHex [idx] := hex (dig - (next * 16));
        idx := idx -1;
        dig := next;
        next := dig div 16;
      end;
   ToHex [idx] := hex (dig);
 end;

begin
Assign(SinFile,'B:SINE.DAT');
Rewrite(SinFile);
x := 0;
f := 0;
i := -90;
s := pi/180;
while i < 91 do
 begin
 k := i + 90;
 x := (i*s);
 f := 1 + sin(x);
 f := 100*f;
 DecToHex (round (f), hexar);
 DecToHex (k,hexcon);
 if i = -90 then
  begin
  writeln ('DEGREES  LOC  100*SIN X + 1    HEXLOC       HEX');
  end;
  write (i:3,'    ',k:3,' ');
  writeln (f:10:5,'  ',hexcon,'   ' ,hexar);
  if i = -90 then
   begin
   writeln(SinFile,'DEGREES 100*SIN(X)+1 HEXLOC 100*SIN(X)+1[HEX]');
   end;
   writeln(SinFile,k:5,'  ',f:10:5, ' ',hexcon,' ',hexar);
   i := i + 1;
 end;
 Close(SinFile);

end.
```

2

PAL32VX10 Uses Buried Register for Input-Intensive State Machine Designs

AN-160B

The PAL®32VX10 offers design engineers several features not previously found in programmable logic devices (PLDs). Many of these features are found in the output macrocell, which allows over thirty possible configurations. This application note will highlight some of these features, and show how they are used in Dual Binary Rate Multipliers.

The PAL32VX10 is a 24-pin PLD with twelve dedicated inputs. The ten I/O macrocells have dual feedback, providing twenty additional array inputs. Each macrocell has the architecture shown in Figure 1.

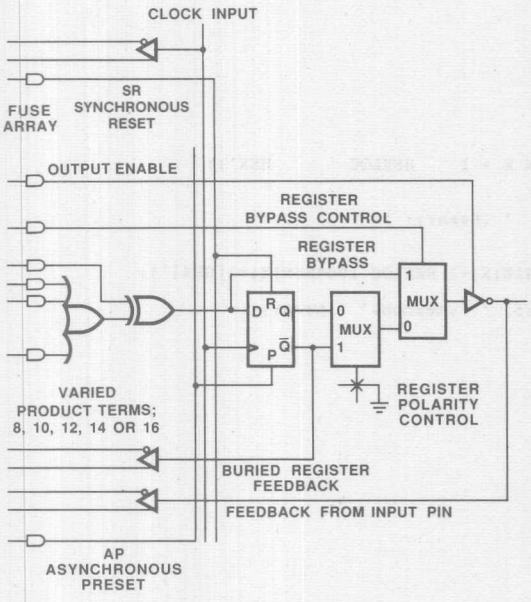


Figure 1. PAL32VX10 Macrocell

Macrocell Description

A D-type flip-flop is clocked by a dedicated input which also feeds the programmable array. The flip-flop data input is fed by a varied number of product terms, from eight to sixteen per output. The sum of these products feeds an Exclusive-OR (XOR) gate controlled by an additional product term. Global reset and preset signals are also controlled by product terms.

A fuse controls flip-flop output polarity. A product term allows the flip-flop to be bypassed, and another product term enables the output. Feedback signals come from the flip-flop and the I/O pin, providing dual feedback.

The device is manufactured with Monolithic Memories' advanced oxide-isolated process, providing high speed and low power consumption. The propagation delay is 30 ns maximum for the standard version and 25 ns maximum for the A-speed version. Power consumption is only 180 mA maximum.

Dual Binary Rate Multipliers

The availability of the clock input in the fuse array avoids the addition of external logic in applications where it is necessary to use the system clock as a logic input. In many synchronous PLDs the system clock input is connected only to the register file. The rising edge of the clock input synchronizes data flow through the D-type registers, but the clock is not available as a Boolean signal in the fuse array.

The application of the Dual Binary Rate Multipliers uses the buried register capability, the XOR function to build counters with a T-type register, and the presence of the clock input in the fuse array. The circuit requires sixteen data inputs. There are two data byte inputs DA0-DA7 and DB0-DB7. The two serial outputs are QA associated with the DA port and QB with the DB port. The clock input is divided by sixteen and is available at the QA or QB outputs multiplied by the binary data inputs DA0-DA3 or DA4-DA7, and DB0-DB3 or DB4-DB7.

The output SELOP (Select Output) selects one half of each A or B byte as the multiplying vector. When SELOP is High the lower half of each input byte is selected to multiply the clock input. When Low, the upper half of each input is used to multiply the clock input.

For example, if data inputs DA4-DA7 and DB4-DB7 were binary encoded as six and three, respectively, then six and three clock pulses would be gated through to the outputs QA and QB respectively while SELOP is Low. When SELOP goes High the data inputs DA0-DA3 and DB0-DB3 are selected as the multipliers. If DA0-DA3 contained the binary encoded input of eight, then eight clock pulses would be gated to QA. At the same time, if DB0-DB3 contained fifteen, fifteen clock pulses would appear at the QB output. These waveforms are shown in the oscilloscope photograph in Figure 2.

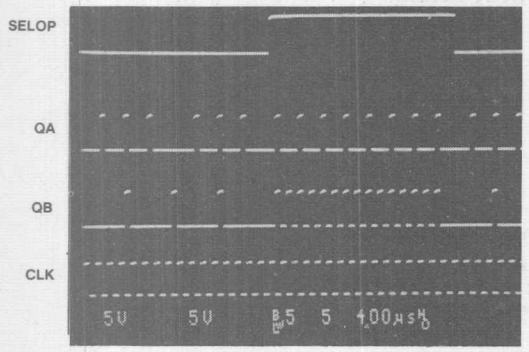
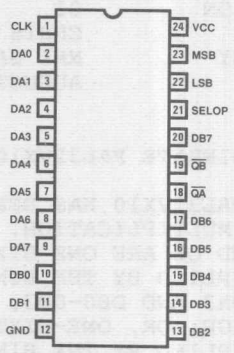


Figure 2. Dual Binary Rate Multipliers Waveforms

The two additional outputs, LSB and MSB, are byte-select outputs. This internal state machine has four unique states, performing a binary count zero to three. Just as the SELOP output can be used to select the multiplying half of each byte, LSB and MSB may be decoded externally to select a one-of-four byte data input to be applied to parallel inputs A or B. A programmable controlled pulse train from four byte inputs can then be available at the QA and QB outputs.

PAL32VX10



Dual Binary Rate Multipliers

Figure 3. Pin Layout

2

```
PATTERN         05
REVISION        01
AUTHOR          CHRIS JAY
COMPANY         MMI SANTA CLARA, CA
DATE            AUGUST 6, 1986
;
;
CHIP BINRATE PAL32VX10
;
;THE PAL32VX10 HAS BEEN PROGRAMMED TO PERFORM BINARY
;RATE MULTIPLICATION.  THE OUTPUTS AT
;QA AND QB ARE ONE-SIXTEENTH OF THE CLOCK INPUT
;MULTIPLIED BY THE BINARY VALUE OF THE INPUTS
;DA0-DA3 AND DB0-DB3, RESPECTIVELY, WHEN SELOP
;IS HIGH; OR, ONE-SIXTEENTH OF THE CLOCK INPUT
;MULTIPLIED BY THE BINARY VALUE OF THE INPUTS DA4-DA7
;AND DB4-DB7, WHEN THE SELOP SIGNAL IS LOW.
;
;PINS    1       2       3       4       5       6
        CLK     DA0     DA1     DA2     DA3     DA4

;PINS    7       8       9       10      11      12
        DA5     DA6     DA7     DB0     DB1     GND

;PINS    13      14      15      16      17      18
        DB2     DB3     DB4     DB5     DB6     QA

;PINS    19      20      21      22      23      24
        QB      DB7     SELOP   LSB     MSB     VCC

GLOBAL  Q0 Q1 Q2 Q3 NC NC NC SEL LSBI MSBI ;BURIED NODES

STRING K1 'Q0* Q1* Q2*/Q3'              ;STRING STATEMENTS
STRING K2 'Q0* Q1*/Q2'                  ;TO DECODE USABLE
STRING K3 'Q0*/Q1'                      ;STATES OF INTERNAL
STRING K4 'Q0*/Q1* Q2* Q3'              ;DIVIDE BY SIXTEEN
STRING K5 'Q0*/Q1*/Q2*/Q3'              ;BINARY COUNTER.
STRING CKNL '/CLK* SEL'                 ;CLOCK SELECTION
STRING CKNH '/CLK*/SEL'                 ;STRING STATEMENTS

EQUATIONS

/Q0     := Q0                           ;CLOCK DIVIDE BY TWO.
                                        ;BURIED REGISTER.

/Q1     := /Q1 :+: Q0                   ;CLOCK DIVIDE BY FOUR.
                                        ;BURIED REGISTER.

/Q2     := /Q2 :+: Q0*Q1                ;CLOCK DIVIDE BY EIGHT.
                                        ;BURIED REGISTER.

/Q3     := /Q3 :+: Q0*Q1*Q2             ;CLOCK DIVIDE BY SIXTEEN.
                                        ;BURIED REGISTER.
```

```
/SEL     := /SEL :+: Q0*Q1*Q2*Q3          ;SELECT O/P HIGH TO SELECT
SELOP    := SEL                           ;DATA 0-3, LOW TO
                                          ;SELECT DATA 4-7.
                                          ;


QA       =   DA0          * K1*CKNL       ;COMBINATORIAL OUTPUT OF
         +   DA3          */Q0*CKNL       ;CLOCK MULTIPLIED BY THE
         +   DA1          * K2*CKNL       ;DA0-DA3 INPUT WHEN SEL
         +   DA2* DA3     * K2*CKNL       ;IS HIGH, AND DA4-DA7
         +   DA2*/DA3     * K3*CKNL       ;WHEN SEL IS LOW.
         +   DA1* DA2     * K3*CKNL       ;
         +  /DA1* DA2* DA3* K4*CKNL       ;
         +  /DA1* DA2* DA3* K5*CKNL       ;
         +   DA4          * K1*CKNH       ;
         +   DA7          */Q0*CKNH       ;
         +   DA5          * K2*CKNH       ;
         +   DA6* DA7     * K2*CKNH       ;
         +   DA6*/DA7     * K3*CKNH       ;
         +   DA5* DA6     * K3*CKNH       ;
         +  /DA5* DA6* DA7* K4*CKNH       ;
         +  /DA5* DA6* DA7* K5*CKNH       ;
QA.CMBF  =  VCC                           ;ENABLE COMBINATORIAL
                                          ;OUTPUT FUNCTION.
                                          ;
QB       =   DB0          * K1*CKNL       ;COMBINATORIAL OUTPUT OF
         +   DB3          */Q0*CKNL       ;CLOCK MULTIPLIED BY THE
         +   DB1          * K2*CKNL       ;DB0-DB3 INPUT WHEN SEL
         +   DB2* DB3     * K2*CKNL       ;IS HIGH, AND DB4-DB7
         +   DB2*/DB3     * K3*CKNL       ;WHEN SEL IS LOW.
         +   DB1* DB2     * K3*CKNL       ;
         +  /DB1* DB2* DB3* K4*CKNL       ;
         +  /DB1* DB2* DB3* K5*CKNL       ;
         +   DB4          * K1*CKNH       ;
         +   DB7          */Q0*CKNH       ;
         +   DB5          * K2*CKNH       ;
         +   DB6* DB7     * K2*CKNH       ;
         +   DB6*/DB7     * K3*CKNH       ;
         +   DB5* DB6     * K3*CKNH       ;
         +  /DB5* DB6* DB7* K4*CKNH       ;
         +  /DB5* DB6* DB7* K5*CKNH       ;
QB.CMBF  =  VCC                           ;ENABLE COMBINATORIAL
                                          ;OUTPUT FUNCTION.
/LSBI    := /LSBI :+: SEL*Q3*Q2*Q1*Q0     ;
LSB      := LSBI                          ;
                                          ;
/MSBI    := /MSBI                         ;
         :+: LSBI*SEL*Q3*Q2*Q1*Q0         ;
MSB      := MSBI                          ;
                                          ;
```

```
SIMULATION                                  ;SIMULATION SECTION.
TRACE_ON        CLK QA QB Q0 Q1 Q2 Q3       ;TRACE ESSENTIAL
                SELOP LSB MSB DA0 DA1       ;WAVEFORMS.
                DA2 DA3 DA4 DA5 DA6 DA7     ;
                DB0 DB1 DB2 DB3 DB4 DB5     ;
                DB6 DB7                     ;
PRLDF   /Q0 /Q1 /Q2 /Q3 SEL /LSBI /MSBI     ;PRELOAD INITIAL VALUES.
SETF    /CLK  /DA0 /DA1 /DA2 /DA3           ;SET INPUT DA0-DA3 = 0.
               DB0 /DB1 /DB2 /DB3           ;SET INPUT DB0-DB3 = 1.
              /DA4  DA5 /DA6 /DA7           ;SET INPUT DA4-DA7 = 2.
               DB4  DB5 /DB6 /DB7           ;SET INPUT DB4-DB7 = 3.
FOR  J :=  0 TO 4 DO                        ;
 BEGIN                                      ;GENERATE FOUR LOOPS.
 IF  J = 1 THEN                             ;ON SECOND LOOP SET
  BEGIN SETF  DA2 DB2 DA6 DB6               ;DA0-DA3 = 4, DB0-DB3 = 5,
  END                                       ;DA4-DA7 = 6, DB4-DA7 = 7.
 IF J = 2 THEN                              ;ON THIRD LOOP SET
  BEGIN SETF  /DA2 DA3 /DB2 DB3             ;DA0-DA3 = 8, DB0-DB3 = 9,
              /DA6 DA7 /DB6 DB7             ;DA4-DA7 = 10, DB4-DB7 = 11.
  END                                       ;ON FOURTH LOOP SET
 IF J = 3 THEN                              ;DA0-DA3 = 12, DB0-DB3 = 13,
  BEGIN SETF   DA2 DB2 DA6 DA6              ;DA4-DA7 = 14, DB4-DB7 = 15.
  END                                       ;
 FOR  I := 1 TO 32 DO                       ;
  BEGIN                                     ;ON EACH LOOP APPLY
  SETF CLK                                  ;32 CLOCK PULSES.
  SETF /CLK                                 ;OUTPUTS QA AND QB
  END                                       ;WILL GENERATE THE
 END                                        ;CLK DIVIDED BY 16 MULTIPLIED
TRACE_OFF                                   ;BY THE BINARY CODE AT THE
                                            ;SELECTED INPUT.
```

Analog to Digital Conversion

It is often necessary to interface analog inputs into a digital system. In digital instruments the signal being measured is usually an analog level. Transducers are used to provide a linear relationship between a measured input signal and a voltage output. An example of a transducer would be a strain gauge. In this case the amplitude of the voltage output would be a measure of mechanical stress. If the measuring device were microprocessor-based or minicomputer-based, the voltage level would have to be converted to a digital binary code before it could be read and processed. An analog to digital converter would be required to accomplish this task.

Conversion Techniques

There are a number of techniques used to provide analog to digital (A to D) conversion. The slowest methods of A to D conversion are based on counters, and would probably be used in instruments such as multimeters. An example would be the self-calibrating Dual Slope Integrator. The fastest A to D converters are based on look-up tables and might be used in video applications. In the case of a Flash Converter high conversion speed is obtained at the expense of a very large number of input comparator circuits. An eight-bit digital output code would require 2^8-1 comparator circuits at the input. The comparator outputs would drive a look-up table from which the closest binary weighting to the analog input would be referenced. Techniques to simplify the input circuitry would reduce the number of comparators required. A Half Flash converter would accomplish this but with some degradation in conversion speed.

Successive Approximation

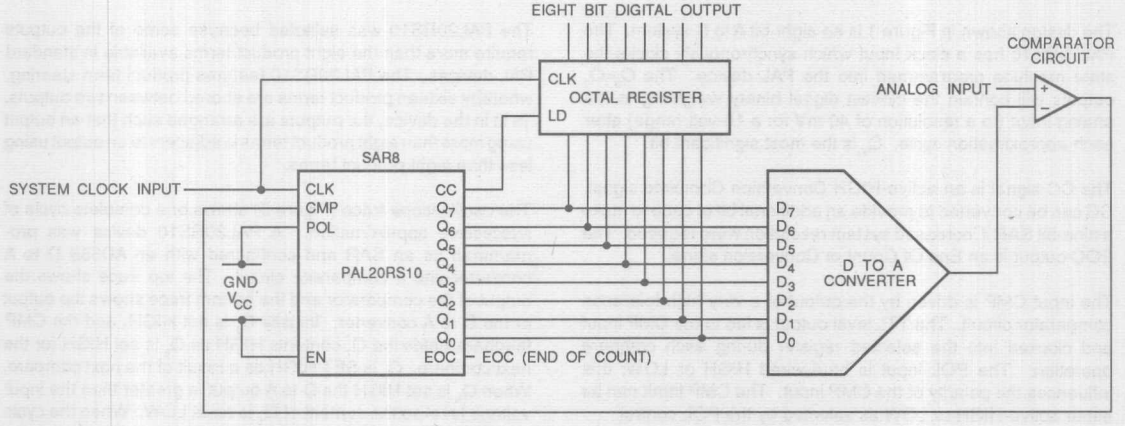
A medium to high conversion speed can be realized with the Successive Approximation technique of conversion, and many circuits exist to perform this type of conversion. Figure 1 shows a diagram of an A to D converter that uses a PAL20RS10 as a Successive Approximation Register. The advantage of using the successive approximation technique of conversion is that the circuit is relatively simple when using a PAL device, and the cycle of conversion is constant. Conversion using counters is no more simple, and does not give a constant conversion time because the time required to produce the closest possible binary output is dependent on the amplitude of the input signal.

The PAL20RS10 has been programmed as a state machine to perform the approximation cycle. The Design Specification title SAR8 (page 2-300) refers to the eight-bit Successive Approximation Register. To provide more bits of resolution, a fourteen-bit SAR14 has been programmed into a PAL32R16 (page 2-303).

Quantization Error

Why would a system require additional bits of data in the SAR circuit? Suppose a two-digit code is used to represent a dynamic voltage range of 0 V to 9 V. A linear mapping would be:

BINARY CODE	00	01	10	11
Input Voltage (Volts)	0.0	3.0	6.0	9.0



601 01

Figure 1. Eight-Bit Successive Approximation Analog to Digital Converter

binary bits to represent any voltages between each separate level. The difference between the actual value and the approximation is the quantization error. The resolution can be increased and quantization error reduced by adding more bits to encode the data. With a three-digit code resolution improves:

BINARY CODE	VOLTAGE LEVELS (VOLTS)
000	0.0000
001	1.2857
010	2.5714
011	3.8571
100	5.1429
101	6.4286
110	7.7143
111	9.0000

However, there remain voltage levels that cannot be accurately represented. In general, if we assume an input dynamic range (Vdr) is to be encoded into equal discrete levels (Vq) by a number (n) of bits of binary data, the relationship becomes:

$Vdr/(2^n - 1) = Vq.$

If a dynamic range of 10 V were represented by eight bits of digital data the input voltage level could be resolved to an accuracy of approximately

$10/(2^8 - 1)$ = approximately 40 mV.

For some systems this might be accurate enough. To decrease quantization error even further the PAL32R16 has been programmed to perform successive approximation with 14 bits of binary information. With the given figures the resolution is

$10/(2^{14} - 1)$ = approximately 0.6 mV.

PAL20RS10 Implementation

The design shown in Figure 1 is an eight-bit A to D system. The PAL20RS10 has a clock input which synchronously clocks the state machine programmed into the PAL device. The Q_0–Q_7 outputs will contain the closest digital binary weighting to the analog input (to a resolution of 40 mV for a 10-volt range) after each approximation cycle. Q_7 is the most significant bit.

The CC signal is an active-HIGH Conversion Complete signal. CC can be converted to provide an additional bit of code to make a nine-bit SAR if increased system resolution were required. The EOC output is an End Of Count or Conversion signal.

The input CMP is driven by the output of a very high tolerance comparator circuit. The TTL level output is fed to the CMP input and clocked into the selected register during each compare operation. The POL input is hard-wired HIGH or LOW; this influences the polarity of the CMP input. The CMP input can be either active-HIGH or LOW as selected by the POL control.

The EN input is the ENable control. If HIGH the SAR is configured

An active-LOW RESET input has been added to the design so a synchronous reset may be applied to the circuit. When LOW, the clock input will set Q_7 HIGH, CC, EOC, and Q_6–Q_0 LOW, shown in Figure 2 at t_1.

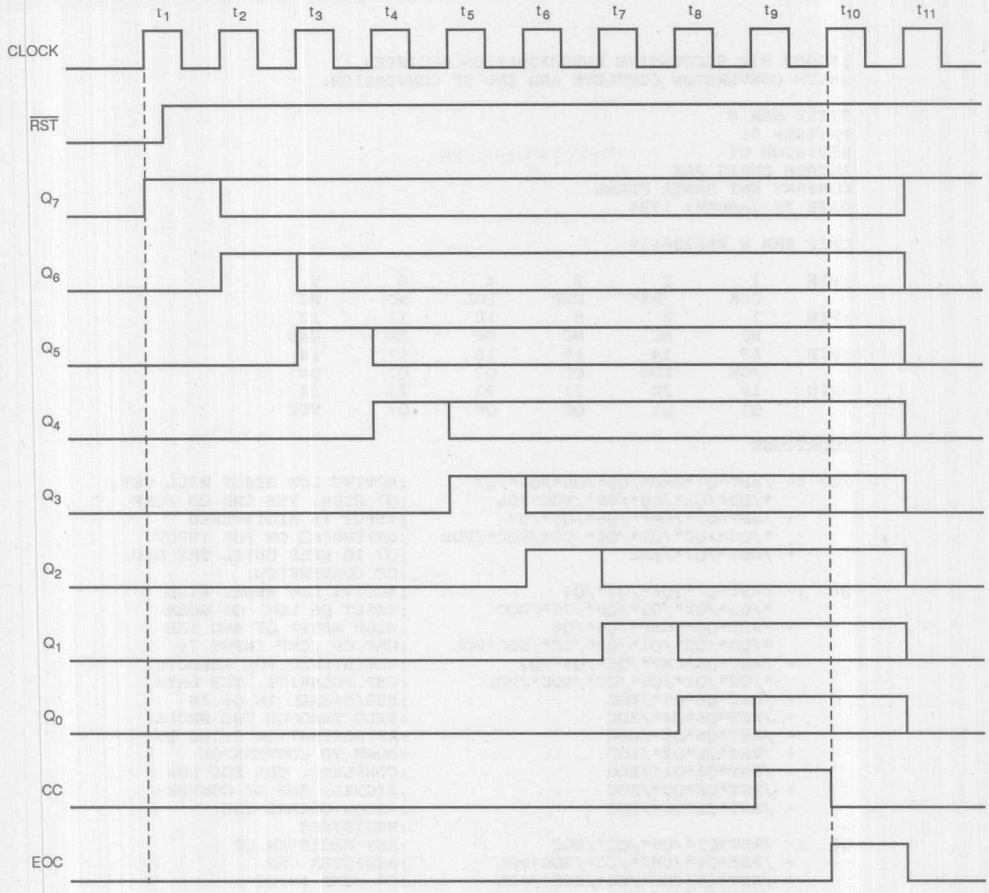
The SAR output (Q_7 HIGH and Q_6–Q_0 LOW) provides a binary input to the D to A converter, generating a corresponding analog signal. The comparator will compare this output with the current value of the analog input. If voltage level of the converted signal is greater than the current level of the analog input the binary weighting of the SAR contents is too large. The feedback from the comparator output to the CMP input will cause the contents of Q_7 to be RESET on the next clock rising edge, t_2 in Figure 2. If the level of the converted signal is less than the input, the binary weighting of the contents of the SAR is not large enough. The comparator feedback will cause the Q_7 register contents to remain SET after the arrival of the clock rising edge t_2. Simultaneously Q_6 will be SET HIGH to perform the next lesser significant approximation.

At t_3 the result of the next CMP input is registered at Q_6 while Q_5 is SET for the next lesser significant compare. The entire cycle is completed with the contents of Q_7–Q_0 containing the closest approximation of an eight-bit digital code to the current analog input. The end of one complete cycle finishes at t_{10} and starts again at t_{11}.

After the CMP input has been loaded into a register, the contents of that register must be held for the duration of the approximation cycle. When the digital code is set up in the register file of the PAL20RS10 it may be synchronously loaded into an octal register, as shown in Figure 1. CC will provide an enabling HIGH (Figure 2) and the digital data is registered at time t_{10} in the cycle. The input to the comparator should be held constant during the conversion cycle in a SAMPLE and HOLD circuit. The EOC signal may be used as an interrupt input to the host processor, which can read the data over an eight-bit data bus by enabling the output of the octal register.

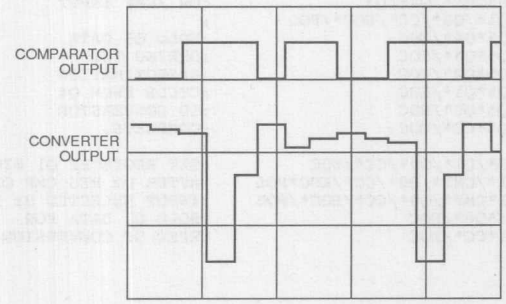
The PAL20RS10 was selected because some of the outputs require more than the eight product terms available in standard PAL devices. The PAL20RS10 features product term steering, whereby sixteen product terms are shared between two outputs. To fit in the device, the outputs are arranged such that an output using more than eight product terms is adjacent to an output using less than eight product terms.

The oscilloscope trace (Figure 3) shows one complete cycle of successive approximation. A PAL20RS10 device was programmed as an SAR and configured with an AD558 D to A converter and a comparator circuit. The top trace shows the output of the comparator and the bottom trace shows the output of the D to A converter. Initially Q_7 is set HIGH, and the CMP feedback holds the Q_7 contents HIGH as Q_6 is set HIGH for the next compare. Q_6 is SET HIGH as a result of the next compare. When Q_5 is set HIGH the D to A output is greater than the input voltage level so the content of Q_5 is reset LOW. When the cycle is complete the result HHLH HLLH is the closest binary weighting to the level of the analog input.



601 02

Figure 2. Eight-Bit Successive Approximation Register Output Waveforms



601 03

Figure 3.

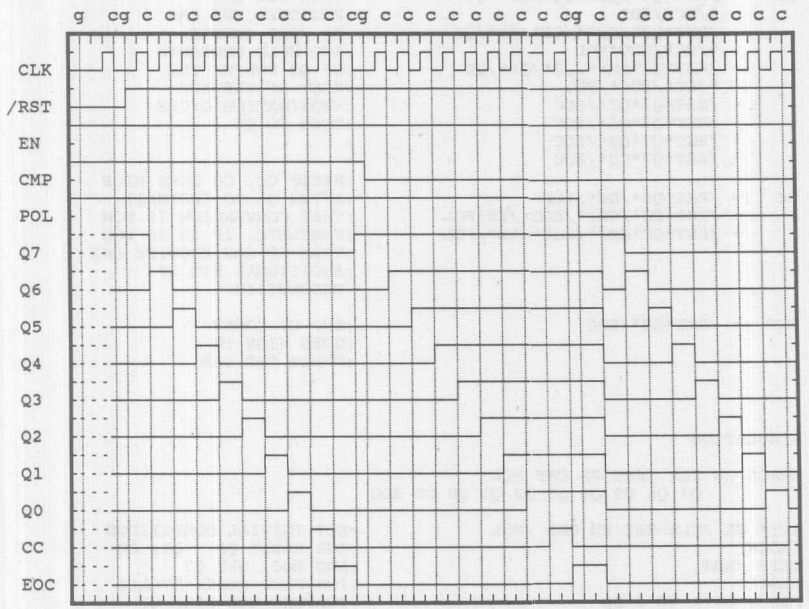
```
;EIGHT BIT SUCCESSIVE APPROXIMATION REGISTER
;WITH CONVERSION COMPLETE AND END OF CONVERSION.

TITLE SAR 8
PATTERN 01
REVISION 01
AUTHOR CHRIS JAY
COMPANY MMI SANTA CLARA
DATE 29 JANUARY 1985

CHIP SAR_8 PAL20RS10

;PIN    1       2       3       4       5       6
        CLK     /RST    CMP     POL     NC      NC
;PIN    7       8       9       10      11      12
        NC      NC      NC      NC      EN      GND
;PIN    13      14      15      16      17      18
        /OE     EOC     CC      Q3      Q2      Q4
;PIN    19      20      21      22      23      24
        Q1      Q5      Q0      Q6      Q7      VCC

EQUATIONS

/Q7  := /RST*Q7*CMP*/Q6*/Q5*/Q4*/Q3          ;ACTIVE LOW RESET WILL SET
        */Q2*/Q1*/Q0*/CC*/EOC*POL            ;Q7 HIGH. THE CMP OR /CMP
      + /RST*Q7*/CMP*/Q6*/Q5*/Q4             ;INPUT IS REGISTERED
        */Q3*/Q2*/Q1*/Q0*/CC*/EOC*/POL       ;DEPENDING ON POL INPUT.
      + /RST*/Q7*/EOC                        ;Q7 IS HELD UNTIL THE END
                                             ;OF CONVERSION.
Q6   := /RST*Q7*/Q6*/Q5*/Q4                  ;ACTIVE LOW RESET WILL
        */Q3*/Q2*/Q1*/Q0*/CC*/EOC            ;RESET Q6 LOW. Q6 GOES
      + /RST*Q6*/CMP*/Q5*/Q4                 ;HIGH AFTER Q7 AND THE
        */Q3*/Q2*/Q1*/Q0*/CC*/EOC*POL        ;CMP OR /CMP INPUT IS
      + /RST*Q6*CMP*/Q5*/Q4*/Q3              ;REGISTERED.POL SELECTS
        */Q2*/Q1*/Q0*/CC*/EOC*/POL           ;CMP POLARITY. THE DATA
      + /RST*Q6*Q5*/EOC                      ;REGISTERED IN Q6 IS
      + /RST*Q6*Q4*/EOC                      ;HELD THROUGH THE WHOLE
      + /RST*Q6*Q3*/EOC                      ;APPROXIMATION CYCLE Q5
      + /RST*Q6*Q2*/EOC                      ;DOWN TO CONVERSION
      + /RST*Q6*Q1*/EOC                      ;COMPLETE. THE EOC LOW
      + /RST*Q6*Q0*/EOC                      ;SIGNAL, END OF CONVER-
      + /RST*Q6*CC*/EOC                      ;-SION CLEARS THE
                                             ;REGISTERS.
Q0   := /RST*Q1*/Q0*/CC*/EOC                 ;SET REGISTER Q0
      + /RST*Q0*/CMP*/CC*/EOC*POL            ;REGISTER CMP
      + /RST*Q0*CMP*/CC*/EOC*/POL            ;OR /CMP INPUT
      + /RST*Q0*CC*/EOC                      ;HOLD Q0 DATA
                                             ;
Q5   := /RST*Q6*/Q5*/Q4*/Q3                  ;SET REGISTER Q5
        */Q2*/Q1*/Q0*/CC*/EOC                ;AFTER Q6. DURING
      + /RST*Q5*/CMP*/Q4*/Q3                 ;COMPARE CYCLE.
        */Q2*/Q1*/Q0*/CC*/EOC*POL            ;REGISTER CMP
      + /RST*Q5*CMP*/Q4*/Q3                  ;OR /CMP INPUT
        */Q2*/Q1*/Q0*/CC*/EOC*/POL           ;
      +  /RST*Q5*Q4*/EOC                     ;HOLD Q5 DATA
      +  /RST*Q5*Q3*/EOC                     ;DURING THE
      +  /RST*Q5*Q2*/EOC                     ;APPROXIMATION
      +  /RST*Q5*Q1*/EOC                     ;CYCLE FROM Q4
      +  /RST*Q5*Q0*/EOC                     ;TO CONVERSION
      +  /RST*Q5*CC*/EOC                     ;COMPLETE.
                                             ;
Q1   := /RST*Q2*/Q1*/Q0*/CC*/EOC             ;SET REGISTER Q1 HIGH
      + /RST*Q1*/CMP*/Q0*/CC*/EOC*POL        ;AFTER Q2 REG CMP OR /CMP
      + /RST*Q1*CMP*/Q0*/CC*/EOC*/POL        ;INPUT SELECTED BY POL
      + /RST*Q1*Q0*/EOC                      ;HOLD Q1 DATA FOR
      + /RST*Q1*CC*/EOC                      ;REST OF CONVERSION
                                             ;
```

```
Q4   := /RST*Q5*/Q4*/Q3*/Q2                ;SET REGISTER Q4
        */Q1*/Q0*/CC*/EOC                  ;AFTER Q5 THEN
      + /RST*Q4*/CMP*/Q3*/Q2               ;COMPARE CYCLE
        */Q1*/Q0*/CC*/EOC*POL              ;REGISTER CMP OR
      + /RST*Q4*CMP*/Q3*/Q2                ;/CMP INPUT.
        */Q1*/Q0*/CC*/EOC*/POL             ;
      + /RST*Q4*Q3*/EOC                    ;HOLD Q4 DATA
      + /RST*Q4*Q2*/EOC                    ;DURING THE
      + /RST*Q4*Q1*/EOC                    ;APPROXIMATION
      + /RST*Q4*Q0*/EOC                    ;CYCLE OF Q3
      + /RST*Q4*CC*/EOC                    ;DOWN TO CC
                                           ;RESET REGISTER Q3
Q2   := /RST*Q3*/Q2*/Q1*/Q0*/CC*/EOC       ;SET REGISTER Q2 HIGH
      + /RST*Q2*/CMP*/Q1*/Q0*/CC*/EOC*POL  ;AFTER Q3 REG CMP OR
      + /RST*Q2*CMP*/Q1*/Q0*/CC*/EOC8/POL  ;/CMP INPUT SELECT W/POL
      + /RST*Q2*Q1*/EOC                    ;HOLD Q2 DATA
      + /RST*Q2*Q0*/EOC                    ;FOR APPROXIMATION
      + /RST*Q2*CC*/EOC                    ;CYCLE TO CC.
                                           ;RESET REGISTER Q3
Q3   := /RST*Q4*/Q3*/Q2*/Q1*/Q0            ;THEN SET Q3.
        */CC*/EOC                          ;REGISTER THE CMP
      + /RST*Q3*/CMP*/Q2*/Q1*/Q0           ;OR /CMP INPUTS
        */CC*/EOC*POL                      ;AND HOLD CONTENTS
      + /RST*Q3*CMP*/Q2*/Q1*/Q0            ;OF Q3 DURING THE
        */CC*/EOC*/POL                     ;REST OF THE APP-
      + /RST*Q3*Q2*/EOC                    ;-ROXIMATION CYCLE
      + /RST*Q3*Q1*/EOC                    ;DOWN TO CC
      + /RST*Q3*Q0*/EOC                    ;
      + /RST*Q3*CC*/EOC                    ;
                                           ;RESET CC. CC GOES HIGH
CC   := /RST*Q0*/CC*/EOC                   ;AFTER Q0 TO INDICATE
      + /RST*CC*/CMP*/EOC*/EN*POL          ;THAT CONVERSION IS NOW
      + /RST*CC*CMP*/EOC*/EN*/POL          ;COMPLETE. IF EN IS LOW
                                           ;THEN CC CAN PROVIDE ONE
                                           ;ADDITIONAL BIT OF
                                           ;RESOLUTION.
                                           ;
EOC  := /RST*CC*/EOC                       ;END OF COUNT
                                           ;GOES HIGH TO
                                           ;RESET THE SAR.


SIMULATION

TRACE_ON CLK /RST EN CMP POL
         Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0 CC EOC

SETF OE /CLK RST EN CMP /POL               ;SET INITIAL CONDITIONS
CLOCKF                                     ;AND RESET Q6 - Q0, CC
SETF /RST                                  ;AND EOC, SET Q7
CLOCKF                                     ;FOR NINE CLOCK CYCLES
FOR I := 1 TO 9 DO                         ;PERFORM ONE PASS OF
BEGIN CLOCKF                               ;SUCCESSIVE APPROXIMATION
END                                        ;WITH CMP HIGH AND POL
SETF /CMP                                  ;LOW. PERFORM SECOND
FOR J := 1 TO 9 DO                         ;PASS WITH CMP LOW
BEGIN CLOCKF CLK                           ;
END                                        ;SET POLARITY PIN ACTIVE
SETF POL                                   ;TEST SUCCESSIVE CYCLE OF
FOR K := 1 TO 9 DO                         ;APPROXIMATION WITH THE
BEGIN CLOCKF CLK                           ;INVERSION INPUT ACTIVE
SETF CMP                                   ;FOR CMP HIGH AND CMP
FOR L := 1 TO 9 DO                         ;LOW
BEGIN CLOCKF CLK                           ;
END                                        ;
CLOCKF                                     ;
END                                        ;
TRACE_OFF                                  ;
```

PALASM SIMULATION, V2.21A MARKET VERSION (07-24-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title	: SAR	Author	: CHRIS JAY
Pattern	: 01	Company	: MMI SANTA CLARA
Revision	: 01	Date	: 29 JANUARY 1985

SAR
Page : 1



```
;FOURTEEN BIT SUCCESSIVE APPROXIMATION REGISTER

TITLE SAR14
PATTERN 01
REVISION 01
AUTHOR CHRIS JAY
COMPANY MMI SANTA CLARA
DATE 29 JANUARY 1985
;
;THE PAL32R16 HAS BEEN DESIGNED AS A FOURTEEN BIT SUCCESSIVE
;APPROXIMATION REGISTER FOR MEDIUM TO HIGH SPEED ANALOG TO
;DIGITAL CONVERSION. THE FOURTEEN OUTPUTS ARE CONNECTED TO
;THE INPUTS OF A FOURTEEN BIT DAC, WITH Q13 AS THE MOST SIG-
;NIFICANT BIT DOWN TO Q0, WHICH IS THE LEAST SIGNIFICANT
;BIT. THE CC OUTPUT IS THE CONVERSION COMPLETE SIGNAL, WHEN
;HIGH IT INDICATES THAT THE CONTENTS OF THE FOURTEEN BIT REGISTER
;HOLDS THE CLOSEST BINARY WEIGHTING TO THE CURRENT VERSION OF
;THE ANALOG SIGNAL UNDERGOING CONVERSION. THE DAC OUTPUT AND
;THE ANALOG INPUTS ARE FED TO A HIGH QUALITY COMPARATOR CIRCUIT,
;THE OUTPUT OF WHICH IS A TTL LEVEL THAT DRIVES THE CMP COMPARE
;INPUT OF THE PAL32R16.
;
CHIP SAR14 PAL32R16

;PIN     1       2       3       4       5       6       7       8
        Q13     NC      Q12     CC      /OE1    /RST    CMP     NC
;PIN     9       10      11      12      13      14      15      16
        NC      VCC     NC      NC      NC      NC      /PLD2   /CLK2
;PIN     17      18      19      20      21      22      23      24
        Q11     Q0      Q10     Q1      Q9      Q2      Q8      Q3
;PIN     25      26      27      28      29      30      31      32
        /OE2    NC      NC      NC      NC      GND     NC      NC
;PIN     33      34      35      36      37      38      39      40
        NC      NC      /PLD1   /CLK1   Q7      Q4      Q6      Q5

STRING H13 'Q13*/CC'              ;HOLD EQUATION FOR Q13
STRING H12 'Q12*/CC*/RST'         ;HOLD EQUATION FOR Q12
STRING H11 'Q11*/CC*/RST'         ;HOLD EQUATION FOR Q11
STRING H10 'Q10*/CC*/RST'         ;HOLD EQUATION FOR Q10
STRING H9  'Q9*/CC*/RST'          ;HOLD EQUATION FOR Q9
STRING H8  'Q8*/CC*/RST'          ;HOLD EQUATION FOR Q8
STRING H7  'Q7*/CC*/RST'          ;HOLD EQUATION FOR Q7
STRING H6  'Q6*/CC*/RST'          ;HOLD EQUATION FOR Q6
STRING H5  'Q5*/CC*/RST'          ;HOLD EQUATION FOR Q5
STRING H4  'Q4*/CC*/RST'          ;HOLD EQUATION FOR Q4
STRING H3  'Q3*/CC*/RST'          ;HOLD EQUATION FOR Q3
STRING H2  'Q2*/CC*/RST'          ;HOLD EQUATION FOR Q2
```

```
EQUATIONS

Q13 :=  Q13*CMP*/Q12*/Q11*/Q10*/Q9*/Q8*/Q7          ;REGISTER CMP I/P
       */Q6*/Q5*/Q4*/Q3*/Q2*/Q1*/Q0*/CC             ;
     +  H13*Q12 + H13*Q11 + H13*Q10                 ;HOLD RESULT OF COMPARE
     +  H13*Q9  + H13*Q8  + H13*Q7                   ;THE APPROXIMATION CYCLE
     +  H13*Q6  + H13*Q5  + H13*Q4                   ;Q12 TO Q0 CLEAR ON RESE
     +  H13*Q3  + H13*Q2  + H13*Q1                   ;CYCLE Q12 TO Q0
     +  H13*Q0  + RST     + CC                       ;AND CONVERSION COMPLETE

Q12 :=  Q13*/Q12*/Q11*/Q10*/Q9*/Q8*/Q7*/Q6          ;CLEAR Q12.DURING
       */Q5*/Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST            ;FIRST APPROXIMATION
     +  Q12*CMP*/Q11*/Q10*/Q9*/Q8*/Q7*/Q6           ;REGISTER COMPARE
       */Q5*/Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST            ;INPUT.
     +  H12*Q11 + H12*Q10 + H12*Q9                  ;HOLD CONVERSION RESULT
     +  H12*Q8  + H12*Q7  + H12*Q6                  ;DURING APPROXIMATION CY
     +  H12*Q5  + H12*Q4  + H12*Q3                  ;Q11 TO Q0 CLEAR ON RESE
     +  H12*Q2  + H12*Q1  + H12*Q0                  ;AND CONVERSION COMPLETE

Q11 :=  Q12*/Q11*/Q10*/Q9*/Q8*/Q7*/Q6*/Q5          ;HOLD Q11 LOW
       */Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST                ;BEFORE COMPARE
     +  Q11*CMP*/Q10*/Q9*/Q8*/Q7*/Q6*/Q5           ;CYCLE. REGISTER
       */Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST                ;COMPARE INPUT.
     +  H11*Q10 + H11*Q9 + H11*Q8                   ;HOLD RESULT OF COMPARE
     +  H11*Q7  + H11*Q6 + H11*Q5                   ;DURING APPROXIMATION
     +  H11*Q4  + H11*Q3 + H11*Q2                   ;CYCLE Q10 TO Q0.  RESET
     +  H11*Q1  + H11*Q0                           ;AND CC CLEARS REGISTER.

Q10 :=  Q11*/Q10*/Q9*/Q8*/Q7*/Q6*/Q5*/Q4           ;HOLD Q10 LOW
       */Q3*/Q2*/Q1*/Q0*/CC*/RST                    ;BEFORE COMPARE
     +  Q10*CMP*/Q9*/Q8*/Q7*/Q6*/Q5*/Q4            ;CYCLE. REGISTER
       */Q3*/Q2*/Q1*/Q0*/CC*/RST                    ;RESULT OF COMPARE
     +  H10*Q9 + H10*Q8 + H10*Q7                    ;HOLD RESULT ON
     +  H10*Q6 + H10*Q5 + H10*Q4                    ;APPROXIMATION
     +  H10*Q3 + H10*Q2 + H10*Q1                    ;CYCLE Q9 TO Q0.
     +  H10*Q0                                      ;RESET AND CC
                                                    ;CLEARS REGISTER.
Q9  :=  Q10*/Q9*/Q8*/Q7*/Q6*/Q5*/Q4*/Q3             ;HOLD Q9 LOW
       */Q2*/Q1*/Q0*/CC*/RST                        ;BEFORE COMPARE.
     +  Q9*CMP*/Q8*/Q7*/Q6*/Q5*/Q4*/Q3             ;REGISTER RESULT
       */Q2*/Q1*/Q0*/CC*/RST                        ;OF COMPARE.
     +  H9*Q8 + H9*Q7 + H9*Q6                       ;HOLD RESULT ON
     +  H9*Q5 + H9*Q4 + H9*Q3                       ;APPROXIMATION
     +  H9*Q2 + H9*Q1                              ;CYCLE Q8 TO Q0.
     +  H9*Q0                                      ;RESET AND CC
                                                    ;CLEARS REGISTER.
Q8  :=  Q9*/Q8*/Q7*/Q6*/Q5*/Q4*/Q3*/Q2             ;HOLD Q8 LOW
       */Q1*/Q0*/CC*/RST                            ;BEFORE COMPARE
     +  Q8*CMP*/Q7*/Q6*/Q5*/Q4*/Q3*/Q2             ;REGISTER RESULT
       */Q1*/Q0*/CC*/RST                            ;OF COMPARE.
     +  H8*Q7 + H8*Q6 + H8*Q5                       ;HOLD RESULT ON
     +  H8*Q4 + H8*Q3 + H8*Q2                       ;APPROXIMATION
     +  H8*Q1 + H8*Q0                              ;CYCLE Q7 TO Q0
Q7  :=  Q8*/Q7*/Q6*/Q5*/Q4*/Q3*/Q2*/Q1             ;HOLD Q7 LOW
       */Q0*/CC*/RST                                ;BEFORE COMPARE
     +  Q7*CMP*/Q6*/Q5*/Q4*/Q3*/Q2*/Q1             ;REGISTER RESULT
       */CC*/RST                                    ;OF COMPARE.
     +  H7*Q6 + H7*Q5 + H7*Q4                       ;HOLD CONTENTS
     +  H7*Q3 + H7*Q2 + H7*Q1                       ;OF RESULT ON
     +  H7*Q0                                      ;APPROXIMATION
                                                    ;CYCLE Q6 TO Q0
```

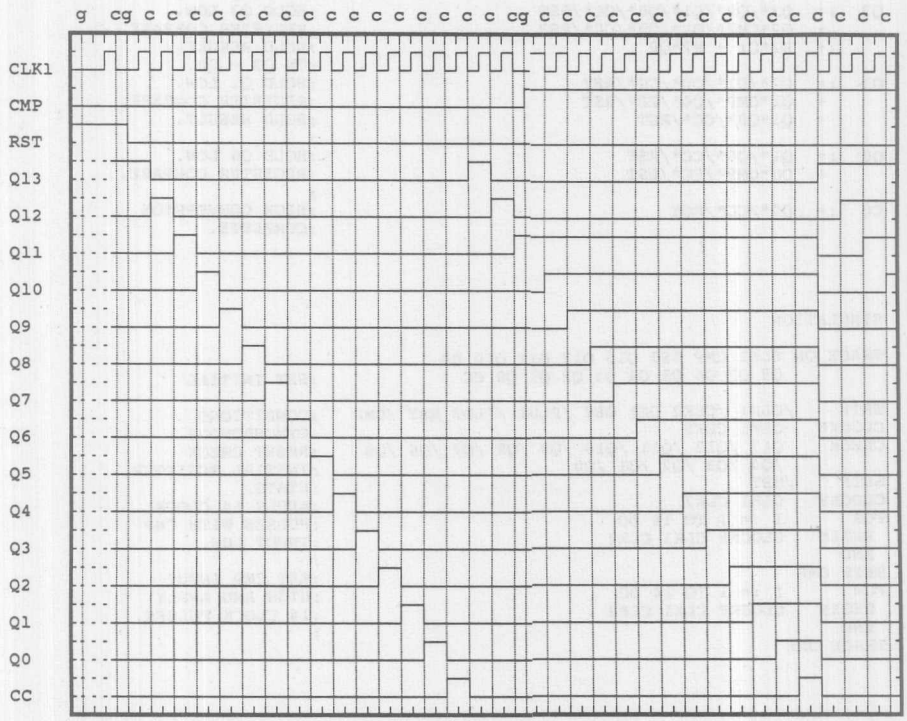
```
Q6   :=  Q7*/Q6*/Q5*/Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST     ;HOLD Q6 LOW BEFORE COMP
     +   Q6*CMP*/Q5*/Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST     ;REGISTER RESULT OF
     +   H6*Q5 + H6*Q4 + H6*Q3                       ;COMAPRE. HOLD RESULT
     +   H6*Q2 + H6*Q1 + H6*Q0                       ;ON APPROXIMATION
                                                     ;Q5 - Q0.
Q5   :=  Q6*/Q5*/Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST        ;HOLD Q5 LOW BEFORE COMP
     +   Q5*CMP*/Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST        ;REGISTER RESULT OF COMP
     +   H5*Q4 + H5*Q3 + H5*Q2                       ;HOLD RESULT ON
     +   H5*Q1 + H5*Q0                               ;APPROXIMATION
                                                     ;CYCLE Q4 - Q0.
Q4   :=  Q5*/Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST            ;HOLD Q4 LOW
     +   Q4*CMP*/Q3*/Q2*/Q1*/Q0*/CC*/RST            ;REGISTER COMPARE
     +   H4*Q3 + H4*Q2 + H4*Q1                       ;HOLD RESULT ON
     +   H4*Q0                                       ;Q3 - Q0.
                                                     ;
Q3   :=  Q4*/Q3*/Q2*/Q1*/Q0*/CC*/RST                 ;HOLD Q3 LOW
     +   Q3*CMP*/Q2*/Q1*/Q0*/CC*/RST                 ;REGISTER COMPARE
     +   H3*Q2 + H3*Q1 + H3*Q0                       ;HOLD RESULT ON
                                                     ;CYCLE Q2 - Q0.
Q2   :=  Q3*/Q2*/Q1*/Q0*/CC*/RST                     ;HOLD Q2 LOW.
     +   Q2*CMP*/Q1*/Q0*/CC*/RST                     ;REGISTER COMPARE
     +   H2*Q1 + H2*Q0                               ;HOLD RESULT
                                                     ;ON Q2 - Q0.
Q1   :=  Q2*/Q1*/Q0*/CC*/RST                         ;HOLD Q1 LOW.
     +   Q1*CMP*/Q0*/CC*/RST                         ;REGISTER COMPARE.
     +   Q1*Q0*/CC*/RST                              ;HOLD RESULT.
                                                     ;
Q0   :=  Q1*/Q0*/CC*/RST                             ;HOLD Q0 LOW.
     +   Q0*CMP*/CC*/RST                             ;REGISTER COMPARE.
                                                     ;
CC   :=  Q0*/CC*/RST                                 ;HIGH CONVERSION
                                                     ;COMPLETE.
```

```
SIMULATION

TRACE_ON CLK1 CMP RST Q13 Q12 Q11 Q10 Q9
         Q8 Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0 CC               ;SET INITIAL

SETF    /CLK1 /CLK2 OE1 OE2 /PLD1 /PLD2 RST /CMP     ;CONDITIONS
CLOCKF   CLK1 CLK2                                   ;SYNCHRONOUS
CHECK    Q13 /Q12 /Q11 /Q10 /Q9 /Q8 /Q7 /Q6 /Q5      ;RESET CHECK
         /Q4 /Q3 /Q2 /Q1 /Q0                         ;INITIAL REGISTER
SETF    /RST                                         ;STATE.
CLOCKF   CLK1 CLK2                                   ;APPLY 14 CLOCK
FOR      I := 1 TO 14 DO                             ;PULSES WITH CMP
 BEGIN   CLOCKF CLK1 CLK2                            ;INPUT LOW.
 END                                                 ;
SETF CMP                                             ;SET CMP INPUT
FOR      I := 1 TO 14 DO                             ;HIGH AND APPLY
 BEGIN   CLOCKF CLK1 CLK2                            ;14 CLOCK PULSES.
 END                                                 ;
TRACE_OFF
```

PALASM SIMULATION, V2.21 P- PROGRAMMER VERSION (21-OCT-1986)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title	: SAR14	Author	: CHRIS JAY
Pattern	: 01	Company	: MMI SANTA CLARA
Revision	: 01	Date	: 29 JANUARY 1985

SAR14
Page : 1



PAL Devices, PROMs, FIFOs, and Multipliers Team up to Implement Single-Board High-Performance Audio Spectrum Analyzer

AN-100

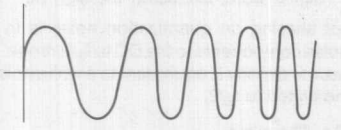
Introduction

This application note illustrates a high-performance audio spectrum analyzer. This circuit can analyze high-fidelity audio signals with a resolution of 20 Hz and an input bandwidth of 20 kHz. It is useful in production test, performance evaluation, or adjustment of high-fidelity audio equipment. The analyzer provides a sweep generator output for rapid analysis of audio filter frequency response.

The design techniques used to implement the analyzer are quite general, and can be applied to a wide variety of DSP tasks. An understanding of the approach used will suggest solutions to a number of DSP problems. The architecture chosen for the spectrum analyzer is controlled by a microprogram stored in PROM. Many other applications can be accommodated by changing the microprogram. The high performance of this architecture provides an attractive price/performance alternative to other DSP approaches.

Spectrum Analyzer Functions

The spectrum analyzer requires many of the functions commonly used in DSP. Figure 1 shows the analyzer functions. An input signal is mixed with a swept audio sinewave oscillator (below).



Frequency Sweep

The frequency sweep acts as a sampler, starting from DC and increasing to its maximum frequency.

Mixing is accomplished by multiplying the input signal by the sinewave. From basic trigonometry:

$$\cos w_1 t \times \cos w_2 t = 1/2 \cos (w_1 + w_2)t + 1/2 \cos (w_1 - w_2)t \quad (1)$$

The mixing process generates two new sinewaves whose frequencies are the sum and difference of the input sinewave frequencies. When the sinewave oscillator matches the frequency of an input signal component, a DC term is generated in proportion to the amplitude of that component:

$$\cos w_1 t \times \cos w_1 t = \cos^2 w_1 t = \mathbf{1} + 2 \cos w_1 t$$

The DC term is extracted by a narrow lowpass filter. Due to the finite bandwidth of this lowpass filter, mixer output signals whose frequencies fall within the filter passband also appear at the filter output. As a result, the analyzer output will represent the energy contained in a range of frequencies, from the sinewave frequency minus the filter cutoff frequency, to the sinewave frequency plus the filter cutoff frequency. The effective bandwidth of the analyzer is twice the lowpass filter bandwidth.

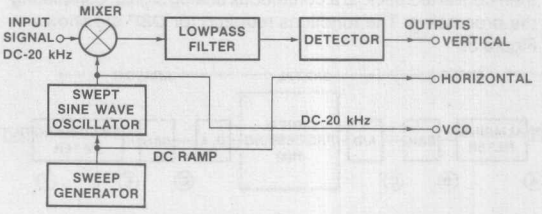


Figure 1. Spectrum Analyzer Functions

A detector converts the lowpass filter output to a DC voltage representing the total energy in the filter passband. If this DC voltage is plotted on a vertical axis with the sinewave oscillator frequency (represented by the sweep voltage) controlling the horizontal axis, a spectrum of the input signal results.

Other mixing schemes can be used to extract the spectrum. However, this "direct conversion" approach has two significant advantages. As shown in Figure 2, the swept oscillator output can be used to plot the frequency response of an audio filter. Other schemes require additional mixing to achieve the same results.

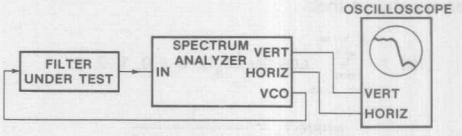


Figure 2. Filter Test Mode Setup

The direct conversion scheme confines the frequencies of all signals following the mixer to the lowpass filter bandwidth. Limiting the signal bandwidth has great benefit when the analyzer is implemented digitally. This benefit can be better understood with a brief review of DSP theory.

Digital Signal Processing Theory Review

Digital signal processing is accomplished by first converting the continuous analog input signal to a series of digital numbers. The digital numbers are then manipulated to perform the required signal processing. The processed digital numbers are then converted back to a continuous analog signal, completing the processing. The functions required for DSP are shown in Figure 3.

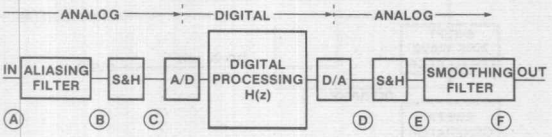


Figure 3. DSP Functions

Sampling

Representing a continuous input signal would require an infinite array of digital numbers. A finite collection of digital numbers can be obtained by considering the signal amplitude at discrete, periodic points in time. This process is called *sampling,* and is equivalent to multiplying the input signal by a periodic train of impulses of unit amplitude. The *sampling theorem* states that the input signal can be reconstructed without distortion if the input is bandlimited to contain no frequency components greater than half the sampling frequency. The sampling theorem means that the discrete samples completely represent the input signal, as long as the bandwidth constraint is met.

Aliasing

What is really happening during the sampling process? Consider the Fourier series representation of a periodic unit impulse train. It can be shown that:

$$f(t) = \sum_{k=-\infty}^{k=\infty} \cos(2\pi k f_s t),\ k = 0, 1, 2, 3, \ldots \qquad (2)$$

$$\text{where } f_s = \frac{1}{\text{sample period}}$$

The periodic impulse train is equivalent to a series of sinusoids consisting of all harmonics of the sampling frequency, including a DC term. Recalling Eq. (1) all possible sum and difference frequencies will be generated when the impluse train and the input signal are multiplied. This process is shown graphically in Figure 4. Observe that if the input contains frequencies greater than half the sampling frequency, the spectra in Figure 4 will overlap. This overlap phenomenon is known as *aliasing distortion,* and introduces noise in the signal.

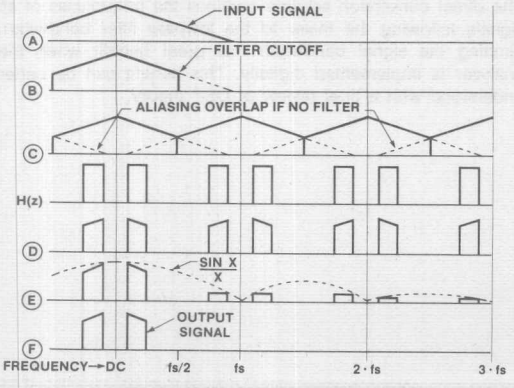


Figure 4. Aliasing Spectra of Figure 3 DSP Functions

Another consequence of the sampling process is that high-frequency signal components near a harmonic of the sampling frequency will be mixed to produce new signal components near DC. These new components have frequencies within the desired signal passband, but are really "alias" high-frequency components. The phenomenon is called *aliasing.*

To eliminate the undesirable effects of aliasing, a continuous analog lowpass filter is placed before the sampler. This *aliasing filter* removes frequency components beyond the $f_s/2$ limit.

Quantizing

The input samples are converted to a series of digital numbers by an analog-to-digital (A/D) converter. The A/D converter operates by *quantizing* the continuous sample amplitude into a finite number of amplitude ranges, and then assigning a digital number to represent the quantized amplitude value. As might be expected, this process introduces noise in the signal, known as *quantization distortion.* The quantization distortion is in the form of a "white" or broadband random noise, whose RMS amplitude is:

$$\sigma^2 = \frac{1}{12} 2^{-2b} \qquad (3)$$

where b is the number of bits in the output digital word, excluding the sign bit

The effect of aliasing on quantization noise is to alias high frequency noise components to the DC to $f_s/2$ range. The resulting noise spectral density is equivalent to a white noise of amplitude σ^2, bandlimited to $f_s/2$.

Dynamic Range

The A/D output contains a finite number of bits. *Dynamic range* is defined as the ratio of the maximum-to-minimum signal amplitude that can be represented by the digital numbers. Dynamic range is determined by the number of bits in the digital numbers, and by the noise "floor."

For a digital number containing b bits plus a sign bit, the dynamic range would be:

$$\text{Dynamic range (dB)} = 10 \log_{10} 2^{-2b} \qquad (4)$$

The noise floor is the sum of all noise components that can appear at the DSP output. The primary noise factors are quantization noise and limit cycle noise (to be discussed shortly). Digital filtering will affect the noise floor by eliminating components of the noise signal. For example, the quantization noise at the DSP output is:

$$N_Q \text{ (dB)} = 10 \log_{10} \left[\sigma^2 \frac{BW}{f_s/2} \right] \qquad (5)$$

where BW is the net bandwidth of the digital filters

The noise components are uncorrelated, and are therefore combined by adding the power of each noise component. Remember that

$$\text{Power (absolute)} = \log_{10}^{-1} \left[\text{Power (dB)}/10 \right] \qquad (6)$$

The resulting dynamic range is:

$$\text{Dynamic range (dB)} = 10 \log_{10} \frac{0.5}{\Sigma \text{ Noise Power}} \quad (7)$$

where 0.5 = the maximum mean-squared amplitude

The overall dynamic range is the lesser of the result given by Eq. (4) or Eq. (7). In a practical system, the width of the digital numbers can vary. The dynamic range is usually calculated for all critical points in a digital system, with the overall dynamic range being the worst case value.

Digital Processing

The digital numbers from the A/D converter are manipulated to process the signal. Carrier generation, filtering, and nonlinear operations are performed by appropriate "number crunching".

Generation of sinusoidal carriers is easily accomplished using a linear ramp function (digital up/down counter) and converting the results to sinusoidal samples using ROM lookup tables. Alternately, recursive equations can produce the desired carriers.

Nonlinear operations on the digital numbers must be handled with care. Since aliasing is always present in the sampled domain, harmonics generated by nonlinear operations can alias to lower frequencies. The aliasing occurs "immediately," since it can be shown that performing a nonlinear operation in the sampled domain is equivalent to first performing the nonlinear operation on a continuous signal and then sampling the result without bandlimiting the sampler input.

The sampling rate can be changed to improve the efficiency of the digital processing. For example, discarding every other digital number would reduce the effective sampling rate by a factor of two. If the processing at the higher sample rate includes digital aliasing filters to remove components greater than half the lower sample rate, the requirements of the sampling theory are still met. The sampling rate can be increased by repeating digital sample values. This repetition is equivalent to a "sample and hold" operation, and modifies the signal spectrum by

$$F'(jw) = F(jw) \times \frac{\sin(wT/2)}{wT/2} \quad (8)$$

where $w = 2\pi \times \text{freq}$

T = input (longer) sample period

The effects of changing the sampling rate are best determined by plotting the resulting aliasing spectra.

Digital Filtering

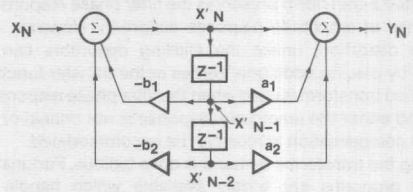
Digital filtering is accomplished using multiplication, addition, and delay. For example, consider the biquadratic filter section in Figure 5. If z^{-1} is defined to be a unit sample period delay operator, then the input-to-output transfer function of the biquadratic section is:

$$H(z) = \frac{1 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}} \quad (9)$$

The biquadric sections can be cascaded to implement higher-order filters.

The Laplace transform of a unit delay is e^{-sT}, where T is the delay period. Remember that z^{-1} represents an inverse operator, so that $z \times z^{-1} = 1$. Thus,

$$z = e^{sT}, \text{ where } s = \alpha + jw \quad (10)$$



$X'_N = X_N - b_1 X'_{N-1} - b_2 X'_{N-2}$

$Y_N = X'_N + a_1 X'_{N-1} + a_2 X'_{N-2}$

Figure 5. Digital Biquadratic Filter Section

Digital filter poles and zeroes (in the z-plane) can be mapped into the s-plane to determine the equivalent analog filter function, and vice-versa. The digital filter section of Eq. (9) corresponds to an analog biquadratic filter section with,

$$H(s) = \frac{s^2 + \alpha_0 w_0 s + w_0^2}{s^2 + \alpha_1 w_1 s + w_1^2} \quad (11)$$

However, the periodic nature of the e^{sT} function causes the digital filter passband to repeat periodically. The effect is the same as aliasing. The analog filter response is mixed with the sampling frequency harmonics to generate the true digital filter response.

Designing Digital Filters

How does one go about designing a digital filter? One approach is to perform a least mean squared error optimization using a computer. The desired function is specified, and the computer adjusts the a_n and b_n values until the desired response is achieved.

A second approach is to design an equivalent analog filter and then convert that design to a digital filter. This approach has great merit, since analog filter design theory is well developed. However, the digital passband will be distorted if the analog equivalent filter has significant response to frequencies greater than $f_s/2$. The aliased passbands overlap at that point.

To circumvent this problem, the analog filter function can be modified to compensate for the aliasing effects. The analog transfer response is modified using several transforms to compensate for aliasing. Unfortunately, the nature of the s-plane to z-plane mapping is such that no transform can compensate for all aliasing effects without introducing other forms of distortion.

The *standard (or impulse invariant) z-transform* represents a direct mapping to the z-plane. No frequency, amplitude, or phase distortion is introduced, but aliasing effects are not compensated. This transform should be used when the analog filter has negligible response to frequencies greater than $f_2/2$.

The *bilinear z-transform* preserves the filter amplitude response in the presence of aliasing. However, the bilinear transform introduces a distortion or *warping* of the frequency axis. As a result, only the filter cutoff frequency can be accurately transformed, using a *pre-warping* technique. Frequencies within the filter passband remain warped, introducing phase distortion in the digital filter response. The bilinear transform is used when the filter amplitude response is more critical than the phase response.

amplitude distortion, unlike the aliasing distortion, can be corrected by placing additional zeroes in the transfer function. The matched transform is used when the filter phase response is critical, and either the amplitude response is not critical or the additional compensation zeroes can be accommodated. Performing the transforms by hand is quite tedious. Fortunately, computer programs are widely available which handle the complete filter synthesis procedure, including z-transforms and pre-warping.

Limit Cycle Noise

An effect of using digital numbers with a finite number of bits is the generation of quantization noise. When implementing digital filters, the quantization noise introduces oscilaltions that are analogous to ringing in analog filters. These oscillations are called *limit cycles*. The limit cycle generates a noise which peaks at frequencies corresponding to the filter pole frequencies. The noise power is roughly proportional to pole Q. Limit cycle noise for a second order filter section of Equation (11) is given by:

$$N_L(\text{dB}) = 10 \log_{10}\left(\frac{2}{12} 2^{-2b} \frac{1+r^2}{1-r^2} \frac{1}{r^4 + 1 - 2r^2 \cos 2w}\right) \qquad (12)$$

where b = number of digital number bits (excluding sign bit)
pole freq. = w_1
pole Q = $1/\alpha_1$

$$w = 2\pi \frac{\text{pole freq.}}{f_s} \qquad r = \exp\left(\frac{-w}{2 \cdot \text{pole Q}}\right)$$

The limit cycle noise must be calculated for each complex pole pair, and adjusted to reflect the response of subsequent filter stages to the limit cycle frequency. Computer programs can calculate limit cycle noise power, including all of these considerations.

Output Signal Reconstruction

Once manipulation of the digital sample numbers is complete, the resulting digital numbers must be converted back to a series of analog output pulses.

A sample-and-hold (S&H) circuit eliminates transients that are introduced during the D/A conversion process. The spectrum of the S&H output is modifed as follows:

$$\text{S\&H } F'(jw) = F(jw) \frac{t}{T} \frac{\sin(wt/2)}{wt/2}$$

where t = hold time T = sample period

An output smoothing filter completes the reconstruction by removing all components with frequencies greater than $f_S/2$. The smoothing filter is often optional, depending on the importance of removing the high-frequency output components.

The spectral effects of reconstruction are shown in Figure 4.

Implementing the Spectrum Analyzer

The architecture used for the spectrum analyzer is shown in Fig. 6. Input signals are digitized and buffered with FIFOs before interface with a common 16-bit data bus. The 16-bit arithmetic unit (AU) provides multiply and accumulate operations. A 16-bit wide RAM stores intermediate results. A 16-bit temporary register facilitates z^{-1} delays and data movement. Outputs are provided using a D/A converter and S&H circuits.

The VCO output is buffered using FIFOs to provide a uniform high-speed sample rate. The VCO output is 12 bits wide, providing a signal-to-quantization noise ratio of 91 dB, using Equations (5) and (7). The calculation assumes a 500-Hz bandwidth. A smoothing filter at the VCO output is not necessary. The filter test configuration of Figure 2 allows the input aliasing filter to remove the effects of VCO high-frequency components, as long as the filter under test is a linear analog circuit.

The vertical and horizontal outputs are intended to interface an oscilloscope or X-Y plotter. The sampling of these outputs can be non-uniform, as long as the outputs track each other. The elastic storage at the input and VCO interfaces permits arbitrary non-uniform processing of the analyzer functions.

The 16-bit resolution of the internal data word provides 90-dB dynamic range according to Equation (4), or 115-dB dynamic range according to Equations (5) and (7), assuming 500-Hz bandwidth and no limit cycle noise and aliasing.

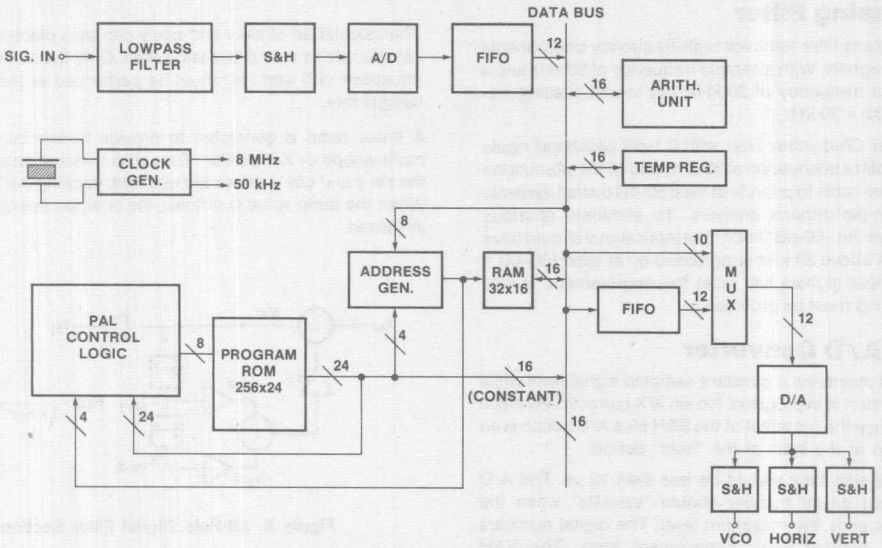


Figure 6. Analyzer Architecture

Microprogram control was selected for the analyzer. PAL devices can efficiently implement sequential state machines. It is possible to encode all control information in PAL devices, but only three packages would be saved (one PROM and two buffers). Distributing the control among several PAL devices would reduce flexibility and make corrections very difficult. The few extra packages required for microprogram control provide an extremely flexible architecture and greatly simplify the PAL device functions.

With the theoretical background and architecture in mind, the spectrum analyzer functions can be defined in detail. The objective is to realize a circuit capable of high-resolution analysis of audio signals in the DC to 20-kHz range. Selectable bandwidth and linear/logarithmic output display are highly desirable. Detailed functions are shown in Figure 7.

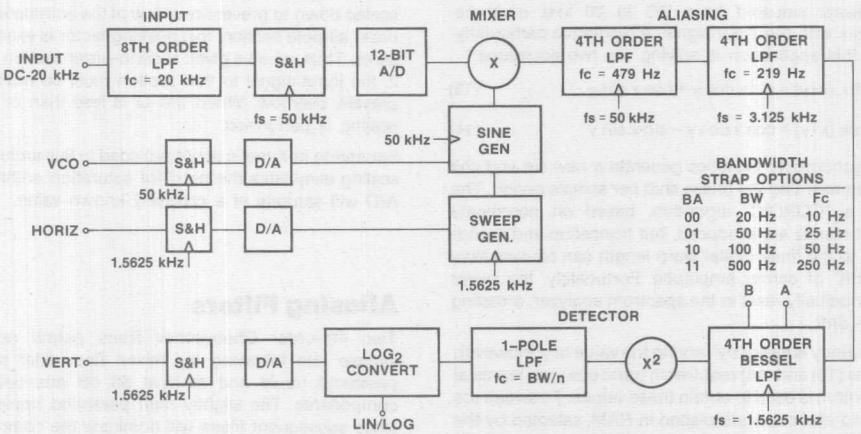


Figure 7. Detailed Functional Diagram

Input Aliasing Filter

An analog lowpass filter removes high-frequency components from the input signals. With a sample frequency of 50 kHz and a maximum input frequency of 20 kHz, the lowest aliasing frequency is (50-20) = 30 kHz.

An eighth-order Chebychev filter with 0.1-dB passband ripple will provide 44 dB of attenuation at 30 kHz, and 86 dB attenuation at 50 kHz. It is desirable to provide at least 60-dB overall dynamic range for high-performace analysis. To eliminate spurious responses above the -60-dB "floor," the input signal should have all components above 30 kHz suppressed by at least (60-44) = 16 dB. Most input signals will meet this requirement. If not, additional filtering must be provided.

S&H and A/D Converter

The input S&H maintains a constant sampled signal level while the A/D conversion is in progress. No sin X/X correction is made for this S&H since the net effect of the S&H plus A/D action is an impulse sample at the start of the "hold" period.

The A/D conversion time should be less than 16 μs. The A/D converter output digital number should "saturate" when the input signal exceeds the maximum level. The digital numbers should be in inverted two's complement form. The S&H acquisition time should be less than 4 μs.

For a full 60-dB overall dynamic range, a 12-bit A/D is required.

Mixer

The mixer multiplies the A/D output by the swept sinewave oscillator value. The multiplication produces sum and difference frequencies, according to equation (1).

Two's complement fractional arithmetic is used throughout the analyzer. Multiplication cannot overflow, since all numbers are less than 1 in magnitude.

Swept Sinusoidal Oscillator (VCO)

A precision swept sinusoid from DC to 20 kHz must be generated to mix with the input signal. A technique particularly well suited to this application is solving the two equations:

$$\sin(x+y) = \sin x \cos y + \cos x \sin y \quad (13)$$

$$\cos(x+y) = \cos x \cos y - \sin x \sin y \quad (14)$$

These two trigonometric identities generate a new sin and cos value with y representing the phase shift per sample period. The technique is a "CORDIC" algorithm, based on coordinate rotation. Exact results are produced, but truncation and round-off errors due to the finite digital word length can cause a slow change or "drift" of carrier amplitude. Fortunately, the swept sinusoid is periodically reset in the spectrum analyzer, arresting this amplitude drift.

The VCO frequency is swept by varying the value of y. However, since equations (13) and (14) require sin y and cos y, an identical CORDIC algorithm is used to obtain these values. To sweep the VCO, then, sinΔ and cosΔ are placed in RAM, selected by the bandwidth setting. These are two fixed numbers originally stored in PROM, and represent the frequency shift per sample time. Equations (13) and (14) are then applied to calculate sin y and cos y, which represent the desired phase shift per sample time. Equations (13) and (14) are executed again to generate the actual sinusoidal output.

The calculation of sin y and cos y can take place at a reduced sample rate to save processing time. Only the last execution of equations (13) and (14) must be performed at the full 50-kHz sample rate.

A linear ramp is generated to provide horizontal drive for an oscilloscope or X-Y plotter. The ramp is incremented each time the sin y and cos y values are updated, tracking the VCO sweep. When the ramp value overflows, the analyzer sweep cycle is re-initialized.

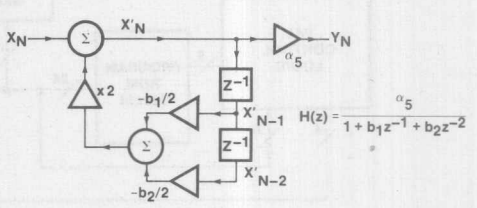


Figure 8. All-Pole Digital Filter Section

Digital Filters

Fig. 8 shows the implementation of the all-pole digital filter sections. Because of the low pole Q values in all filters, the second order sections can be simply cascaded to implement high-order filters. Fig. 8 shows a technique for handling coefficients greater than 1 with fractional number representation.

Scaling must be performed to ensure maximum dynamic range. Filter sections with high-Q poles will show peaking of signals near the pole frequencies. The input to such sections must be scaled down to prevent overflow of the arithmetic. For a second-order all-pole section, this peaking factor is exactly the Q of the poles. Thus, when a given second-order section has a pole Q of 2, the input signal to that section must be multiplied by 0.5 to prevent overflow. When the Q is less than or equal to 1, no scaling is performed.

Saturation arithmetic is not provided in this architecture. Careful scaling eliminates the need for saturation arithmetic, since the A/D will saturate at a precisely known value.

Aliasing Filters

Two 4th-order Chebyschev filters permit reduction of the sample rate following the mixer. Each filter provides 0.3 dB passband ripple and at least 68 dB attenuation of aliasing components. The slightly high passband ripple is acceptable, since subsequent filters will dominate the composite passband shape.

The first filter permits a sample rate reduction factor of 16. It is designed with a passband cutoff frequency of 479 Hz and a sample rate of 50 kHz. Eq. (12) predicts the limit cycle noise for this filter to be –58 dB.

The second filter permits a second sample rate reduction factor of two. Its cutoff frequency is 219 Hz, with a sample rate of 3.125 kHz. Equation (12) predits the limit cycle noise for this filter to be -83 dB. This filter also provides an additional 14-dB attenuation of the limit cycle noise generated in the first aliasing filter, reducing the limit cycle noise from the first filter to -72 dB.

These two filters permit an overall f_S reduction factor of 32 before processing the Bessell filter, detector, and linear-to-$\log_2$ functions. This results in a very substantial throughput improvement. Net execution time is determined by the time to execute a given function multiplied by the sample rate for that function. Thus 32 instructions at the reduced rate will increase the net execution time by an amount equivalent to only 1 instruction at the full sample rate.

Fig. 9 shows the aliasing spectra of the sample rate reduction process.

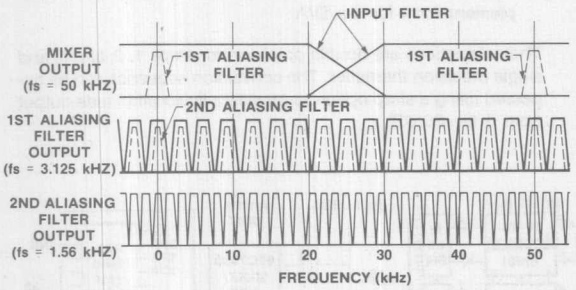


Figure 9. Aliasing Spectra for f_S Reduction

Lowpass Filter

A 4th-order Bessel lowpass filter determines the overall bandwidth of the analyzer. The overall bandwidth is twice the bandwidth of this filter. Overall bandwidths of 20, 50, 100 or 500 Hz are provided by loading the proper set of filter coefficients into RAM when the bandwidth is selected.

The Bessel filter provides an optimal transient response for the analyzer. Good transient response is important, especially at narrow bandwidths, since the spectral peaks are swept with respect to the fiter passband. The net effect is similar to pulsing the filter input. Because the phase response is critical, the matched-z transform is used to convert the analog Bessel design to the z-domain.

The second aliasing filter provides 3 dB of attenuation at 250 Hz. When cascaded with the Bessel filter, which also provides 3 dB attenuation at 250 Hz in the 500 Hz bandwidth mode, the response at this bandwidth is modified. However, since the overall bandwidth is relatively broad, good transient response is still achieved. Cascading these two filters provides a "transitional" filter with a Bessel response at low attenuation and a Chebyschev response at high attenuation. At bandwidths less than 500 Hz, the combination produces an optimal tradeoff between transient response and resolution.

Analysis of Equation (12) reveals that limit cycle noise increases exponentially as the pole frequency is reduced. Operating the lowpass filter at the lowest possible sampling frequency (1.5625 kHz) minimizes limit cycle noise, in addition to improving throughput. Limit cycle noise for the lowpass filter will be -95 dB at the 500-Hz bandwidth, increasing to -67 dB at the 20-Hz bandwidth.

Detector

A square-law detector provides a DC signal corresponding to the energy at the lowpass filter output. From trigonometry:

$$(A \cos wt)^2 = \frac{A^2}{2} (1 + \cos 2wt) \qquad (15)$$

The detector output contains the desired DC term and a single undesired term at frequency 2w. If the square law is ideal (easy in the digital domain), no additional terms are produced. The elimination of harmonics ensures the accuracy of the detector with $f_S/32$ = 1.5625 kHz. The highest component is always less than $f_S/64$ with a 250 Hz maximum lowpass filter cutoff frequency. However, 2w can be anywhere from DC to 500 Hz as the VCO sweeps past the spectral component.

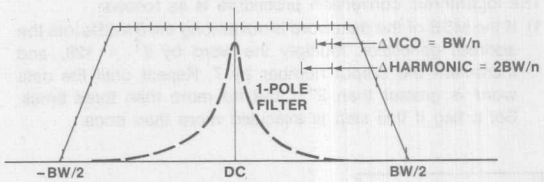


Figure 10. Detector Sweep Filtering

Fig. 10 illustrates a technique to render the effects of the 2w terms negligible. The analyzer passband is divided into n equal intervals. The VCO sweep rate is controlled so that the VCO sweeps BW/n per $f_S/32$ interval. The detector is followed by a single-pole lowpass filter with a 3 dB frequency of BW/n. As the VCO sweeps a component through the passband, the DC term is present in all n intervals, but the 2w term can affect only one interval. The worst-case DC error is 1/n for an ideal cutoff, and is multiplied by $(2^0 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-5.5} + 2^{-6} + 2^{-6.5} + 2^{-7} + 2^{-7.25} + \ldots.) = 1.85$ due to the finite 6 dB/octave rolloff of the single-pole filter. Further analysis reveals that:

$$n = \frac{1.85}{10^{e/10} - 1} \qquad (16)$$

where e represents the resulting error in dB. For e = 0.1 dB, n = 80.

In the filter test mode, the signal frequency and VCO frequency are the same, forcing w = 0. The detector has no error in this mode, but has a 3 dB gain due to the second DC term.

The detector output represents signal energy. Each bit in the detector output word thus represents only 3 dB, and 21 bits are required to reflect a 60 dB dynamic range. Double precision arithmetic is required for the detector ouput and the single-pole filter. The 67C7560 multiplier will handle double precision calculations with a time penalty. Fortunately, the calculations to be performed are simple and the operations take place at the minimum sample rate, reducing the impact on throughput.

Linear-to-Log Conversion

The architecture of Figure 6 is customized to provide an efficient algorithm for linear-to-logarithmic output conversion. The RAM address generator monitors the 8 MSBs of the data bus, and can provide a number indicating the MSB position of a positive

This value is used to scale the data word to quickly left-justify the MSB. A second 4-point lookup table is then used to improve the accuracy of the resulting $\log_2$ conversion. A $\log_2$ conversion is adequate, since:

$$\log_{10} x = \frac{\log_2 x}{\log_2 10} \qquad (17)$$

Equation (17) demonstrates that the output can be displayed in decibels by setting the oscilloscope or X-Y plotter Y-axis gain to the proper value.

Two lookup tables provide .027 dB accuracy for output values from 0 to –45 dB, and 3 dB accuracy from –45 to –84 dB. The logarithmic accuracy is limited by the 10-bit output word length to the D/A. This output can represent 0 to –84 dB in .082 dB increments. The accuracy of the 4-point lookup is therefore sufficient.

The logarithmic conversion procedure is as follows:

1) If the MSB of the data word is not among the 8 MSBs into the address generator, multiply the word by $2^7 = 128$, and increment the output number by 7. Repeat until the data word is greater than 2^{-7}, but no more than three times. Set a flag if this step is executed more than once.

2) Add $\log_2$ of the scale factor to the output word. The conversion is now accurate to 3 dB.

3) If the flag was not set during step 1, multiply the data word by the scale factor to left-justify the MSB position.

4) If the flag was not set during step 1, retrieve an intercept and slope value from the 8-word lookup table (four pairs available). Perform a linear interpolation using:

$$x' = a\,x + b$$

where a is the slope value
b is the intercept value (18)

The conversion is now accurate to .027 dB.

5) Scale the result to provide 84-dB output range with a 10-bit word.

6) Subtract 2^{-1} from the output to convert it to two's complement form for the D/A.

The calculations are double precision for steps 1, 2, and 3, and single precision thereafter. The conversion sequence can be bypassed using a strap option to provide a linear amplitude output from 0 to –30 dB.

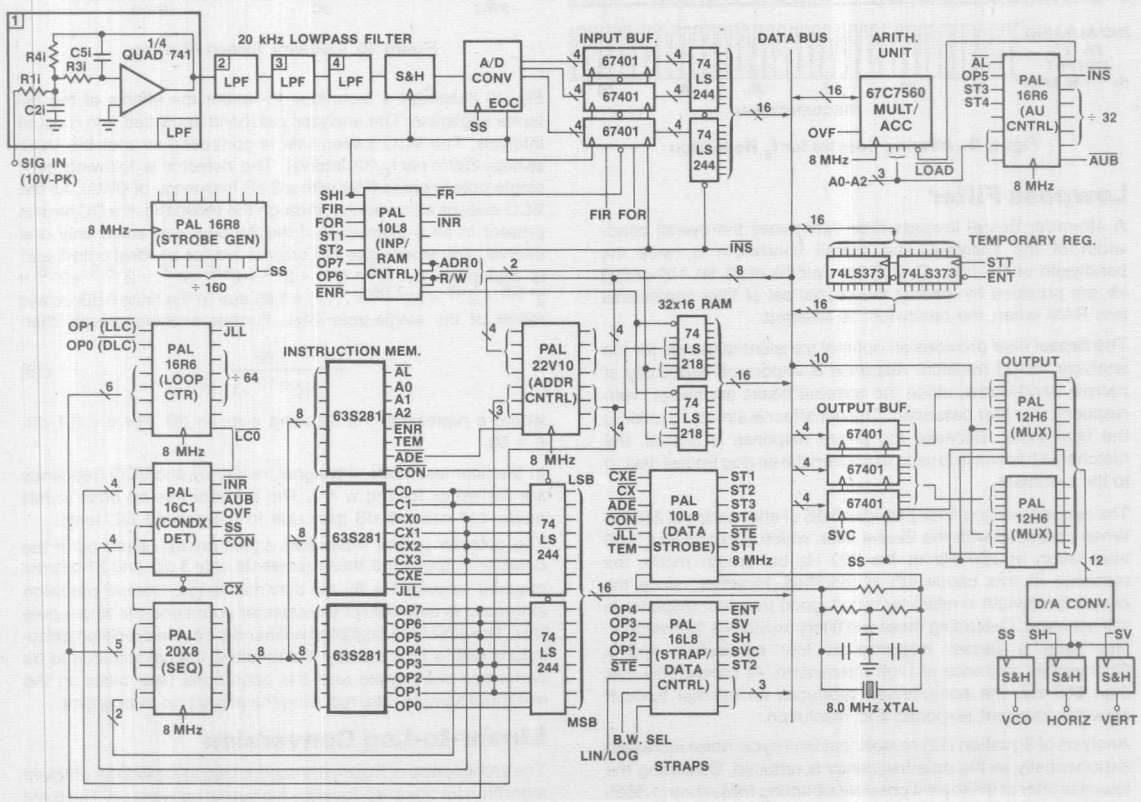


Figure 11. Simplified Schematic Hi-Fi Audio Spectrum Analyzer

Control Logic

Figure 11 shows a simplified schematic of the analyzer. All critical components are shown. Bypass capacitors and some component input connections are omitted for clarity.

The microprogram is stored in three 63S281 PROMs. The microcode word formats are shown in Figure 12. A wide, highly-parallel instruction word ensures maximum flexibility and program efficiency.

Eight PAL devices interpret the instruction word and control the analyzer. Two additional PAL devices generate a 50-kHz strobe from the 8-MHz master clock, and implement the output D/A multiplexer. The control PAL devices function as follows:

Sequencer: A PAL20X8 implements an 8-bit instruction sequencer. The sequencer performs the following operations:

$\overline{C1}$	$\overline{C0}$	$\overline{CX}$	Operation
0	0	X	Increment by 1 (execute next instruction)
0	1	0	Increment by 2 (skip next instruction)
1	0	0	Jump to address
1	1	0	No increment (repeat current instruction)

The $\overline{CX}$ input conditions the sequencer. Conditional branches or skip operations can be implemented. The sequencer will increment if the conditional requirement is not met.

Condition detector: A PAL16C1 monitors up to twelve status flags, and generates $\overline{CX}$. The microcode word includes a 4-bit

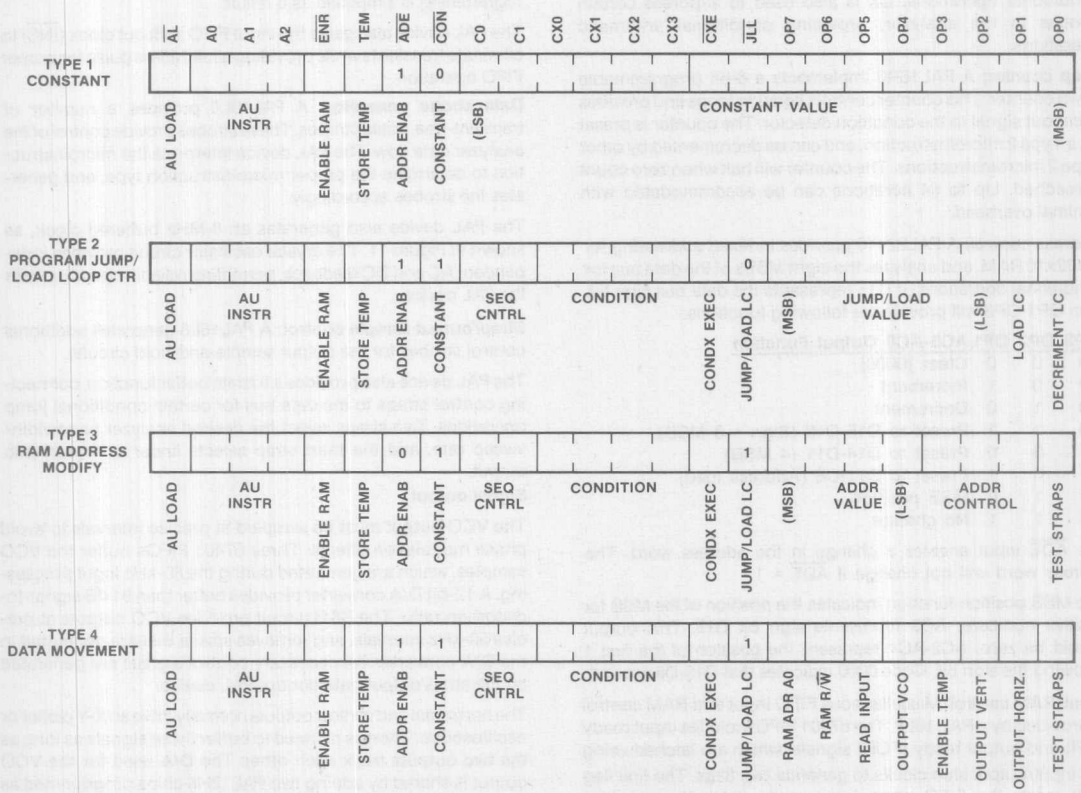


Figure 12. Microinstruction Word Formats

condition word, CX0 through CX3. $\overline{CX}$ will be zero under the following conditions:

CX3	CX2	CX1	CX0	Condition for $\overline{CX}$ = 0
0	0	0	0	Always (unconditional operation)
0	0	0	1	Sample strobe (SS) = 1
0	0	1	0	AU overflow (OVF) = 1
0	0	1	1	AU busy (AUB) = 1
0	1	0	0	Input sample ready (INR) = 0
0	1	0	1	Loop counter timeout ($\overline{LLC}$) = 0
0	1	1	0	$\overline{LLC}$ = 1
0	1	1	1	Address control (AC3-AC0) = 0
1	0	0	0	AC0 = 0
1	0	0	1	AC0 = 1
1	0	1	0	AC1 = 0
1	0	1	1	AC1 = 1
1	1	0	0	AC2 = 0
1	1	0	1	AC2 = 1
1	1	1	0	AC3 = 0
1	1	1	1	AC3 = 1

The assignment is made using the flexible PAL device coding, and is optimized for the analyzer. The user can select a different set of conditions by reprogramming the PAL device.

When the microcode represents a constant (Type 1 microinstruction — see Figure 12) the $\overline{CON}$ input forces $\overline{CX}$ = 1 to suppress conditional operations. $\overline{CX}$ is also used to suppress certain strobes in the analyzer, providing conditional arithmetic operations.

Loop counter: A PAL16R6 implements a 6-bit programmable down counter. This counter controls iteration loops and provides a timeout signal to the condition detector. The counter is preset via a Type 2 microinstruction, and can be decremented by other Type 2 microinstructions. The counter will halt when zero count is reached. Up to 64 iterations can be accommodated with minimal overhead.

Address control: A PAL22V10 provides indexed addressing for the 32x16 RAM, and analyzes the eight MSBs of the data bus for conditional operations. If D15 represents the data bus sign bit, then OP1-OP3 will provide the following functions:

OP3	OP2	OP1	AC3-AC0 Output Function
0	0	0	Clear (0000)
0	0	1	Increment
0	1	0	Decrement
0	1	1	Preset to D15-D12 (Sign + 3 MSB)
1	0	0	Preset to D14-D11 (4 MSB)
1	0	1	Preset to D11-D8 (Address load)
1	1	0	MSB position
1	1	1	No change

The $\overline{ADE}$ input enables a change in the address word. The address word will not change if $\overline{ADE}$ = 1.

The MSB position function indicates the position of the MSB for positive numbers. AC3 represents sign bit D15. This output should be zero. AC2-AC1 represent the position of the first 1 following the sign bit. Code 0000 indicates that D15-D8 are all 0.

Input/RAM control: Miscellaneous FIFO input and RAM control is provided by a PAL 10L8. The 67401 FIFO includes input ready (FIR) and output ready (FOR) signals, which are latched using the input/output shift clocks to generate two flags. The first flag ($\overline{FR}$) resets the FIFO when input ready (latched) goes low, indicating the FIFO capacity is exhausted. The latched output ready signal flag represents input sample ready ($\overline{INR}$). The INR flag is used as a sequencer condition to synchronize wait loops. Use of the FR and INR flags maintains proper fill of the FIFO.

The RAM address LSB (A0) and read-write line (R/$\overline{W}$) are decoded and latched. These signals are provided directly by Type IV microinstructions.

Notice that a clocked register function requires two PAL combinatorial outputs per bit, while a transparent latch function requires only one PAL output per bit.

Arithmetic unit control: The variety of functions listed in Table 3 indicate the utility of the arithmetic unit (AU). A PAL16R6 provides simplified control of the AU.

The PAL devices and AU load signal provide conditional arithmetic operations. Gating the load input will suppress the start of a new arithmetic operation. When $\overline{CXE}$ is high, the operation is performed unconditionally. When $\overline{CXE}$ is low, the operation is performed only if $\overline{CX}$ is low. Combining conditional jumps and conditional AU operations provides a high degree of program flexibility.

The PAL device monitors the AU instructions and generates a busy signal ($\overline{AUB}$). A counter in the PAL device keeps track of variable-length operations to provide the correct output for any instruction sequence. The $\overline{AUB}$ signal conditions the sequencer to synchronize the microprogram to the AU operation. Microprogramming is simplified as a result.

The PAL device also gates the input FIFO shift out clock ($\overline{INS}$) to eliminate transients while providing a full 125-ns pulse for proper FIFO operation.

Data strobe generator: A PAL10L8 provides a number of transient-free, gated strobes. These strobes provide control of the analyzer data flow. The PAL device interprets the microinstruction to determine the proper microinstruction type, and generates the strobes accordingly.

The PAL device also generates an 8-MHz buffered clock, as shown in Figure 11. The crystal oscillator circuit provides independent AC and DC feedback, permitting reliable operation with the PAL device.

Strap/output sample control: A PAL16L8 generates additional control strobes for the output sample-and-hold circuits.

The PAL device also provides a tristate buffer function, connecting control straps to the data bus for certain conditional jump operations. Two straps select the desired analyzer bandwidth/sweep rate, and the third strap selects linear or logarithmic output.

Signal output

The VCO output must be sampled at precise intervals to avoid phase modulation effects. Three 67401 FIFOs buffer the VCO samples, which are generated during the 50-kHz input processing. A 12-bit D/A converter provides better than 91 dB signal-to-distortion ratio. The S&H circuit provides VCO outputs at precise 50-kHz intervals, and removes spikes that are generated in the D/A converter. All necessary control signals are generated by the strap/output data control PAL device.

The horizontal and vertical outputs normally drive an X-Y plotter or oscilloscope. There is no need to buffer these signals as long as the two outputs track each other. The D/A used for the VCO output is shared by adding two PAL12H6 chips programmed as multiplexers. Use of PAL devices requires fewer packages than a TTL multiplexer. Additional S&H circuits decode the multiplexed D/A output to separate the output signals.

Microprogram

The architecture can implement a variety of DSP functions. A microprogram, stored in 63S281 PROMs, customizes the architecture to perform the spectrum analyzer tasks. The microinstruction formats were summarized in Figure 12. The algorithms to be implemented were discussed in the previous section. The step-by-step implementation of these algorithms is converted to a sequence of microinstructions to form the microprogram. The procedure is analogous to programming a microprocessor.

Operation of the microprogram is better understood by considering the allocation of the 74S218 RAM locations, shown in Figure 13. The microprogram consists of two parts. High-speed input processing provides the carrier generation, mixing, aliasing filter and lowpass filter functions. Figure 13a shows the RAM allocation during input processing. The input segment includes an efficient iteration loop, using the PAL device loop counter, to process the 50-kHz functions. The carrier frequency shift and lowpass filter functions are processed at the $f_S/32$ reduced sample rate for maximum throughput efficiency.

The values of sin Δ, cos Δ, and the Bessell filter coefficients depend on the analyzer bandwidth strap selection. These values are stored in a "table" area in Fig. 13, and can be easily changed. The fixed aliasing filter coefficients are stored as constants in the microprogram itself.

Once the input processing is complete, coefficients located in the table area can be changed. This area is re-used by the output program segment to hold the scale factors for the linear-to-log conversion routine. The detector functions are processed, and the logarithmic conversion is started with the RAM allocation of Figure 13(b). The table area is then reloaded with the interpolation coefficients (Figure 13(c)) to complete the logarithmic conversion. Shaded areas in Figure 13 provide temporary data and flag storage for the routines.

The microprogram samples the strap settings and loads the table area with the appropriate coefficients for input processing. The detector filter coefficient (b_1) is also determined and loaded. The input processing is then repeated. This sequence repeats indefinitely. The coefficient loading technique makes efficient use of RAM capacity while eliminating elaborate jump sequences. All coefficient table updates are processed at the minimum sample rate for best efficiency.

The PAL device controllers simplify the microprogram. A PAL device provides hardware iteration loops. The AU controller eliminates wasteful "NO-OP" instructions otherwise required to allow completion of AU operations. The input control PAL device simplifies handshaking with the input logic. With the benefit of the PAL device controllers, the analyzer microprogram easily fits into the 256-instruction capacity of the PROMs.

2

(A4-A1)

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	(A0)
–	–	SIN X	SIN Y	SIN Δ	b1	b1	–	X_{N-1}	X_{N-2}	X_{N-1}	X_{N-2}	X_{N-1}	X_{N-2}	X_{N-1}	X_{N-2}	0
–	–	COS X	COS Y	COS Δ	b2	b2	–	X_{N-1}	X_{N-2}	X_{N-1}	X_{N-2}	X	X	X	VCO RAMP	1
		CARRIER GENERATOR			LOWPASS FILTER			ALIASING FILTERS					LOWPASS FILTER			

(a) INPUT

TEMP LSB	FLAG	X	X	SCALE A	SCALE B	SCALE C	SCALE D	X	X	X	X	X	X	X	X	0
TEMP MSB	OUTPUT	X	X	LOG A	LOG B	LOG C	LOG D	X	X	X	X	X_{N-1} LSB	X_{N-1} MSB	b1	X	1
UTILITY				LIN-LOG SCALING								DETECTOR				

(b) OUTPUT-1

TEMP LSB	FLAG	X	X	a_1	a_2	a_3	a_4	X	X	X	X	X	X	X	X	0
TEMP MSB	OUTPUT	X	X	b_1	b_2	b_3	b_4	X	X	X	X	X	X	X	X	1
UTILITY				LIN-LOG INTERPOLATION												

KEY: ▨ SCRATCH, – NOT USED, X NOT AVAILABLE, □ TABLE

(c) OUTPUT-2

Figure 13. RAM Allocation

```
TITLE           AN-100 DSP Counter
PATTERN         DSPCOUNT
REVISION        1
AUTHOR          Marc Baker
COMPANY         Monolithic Memories
DATE            August 20, 1987

CHIP DSPCOUNT PAL20X8

CLK NC OP3 OP4 OP5 OP6 OP7 C0 C1 /CX NC    GND
/OE NC A7  A6  A5  A4  A3  A2 A1 A0  /LOAD VCC

EQUATIONS

LOAD =  /C0* C1* CX                          ;FED BACK TO EQUATIONS

/A0  :=  /LOAD*/A0                           ;HOLD/INCREMENT BY 2
      +  /C0* C1* CX                         ;LOAD 0
     :+: /C0*/C1                             ;INCREMENT BY 1

/A1  :=  /LOAD*/A1                           ;HOLD
      +  /C0* C1* CX                         ;LOAD 0
     :+: /C0*/C1*     A0                     ;INCREMENT BY 1
      +   C0*/C1* CX*/A0                     ;INCREMENT BY 2

/A2  :=  /LOAD*/A2                           ;HOLD
      +  /C0* C1* CX                         ;LOAD 0
     :+: /C0*/C1*     A1* A0                 ;INCREMENT BY 1
      +   C0*/C1* CX* A1*/A0                 ;INCREMENT BY 2

/A3  :=  /LOAD*/A3                           ;HOLD
      +  /C0* C1* CX*/OP3                    ;LOAD
     :+: /C0*/C1*     A2* A1* A0             ;INCREMENT BY 1
      +   C0*/C1* CX* A2* A1*/A0             ;INCREMENT BY 2

/A4  :=  /LOAD*/A4                           ;HOLD
      +  /C0* C1* CX*/OP4                    ;LOAD
     :+: /C0*/C1*     A3* A2* A1* A0         ;INCREMENT BY 1
      +   C0*/C1* CX* A3* A2* A1*/A0         ;INCREMENT BY 2

/A5  :=  /LOAD*/A5                           ;HOLD
      +  /C0* C1* CX*/OP5                    ;LOAD
     :+: /C0*/C1*     A4* A3* A2* A1* A0     ;INCREMENT BY 1
      +   C0*/C1* CX* A4* A3* A2* A1*/A0     ;INCREMENT BY 2

/A6  :=  /LOAD*/A6                                   ;HOLD
      +  /C0* C1* CX*/OP6                            ;LOAD
     :+: /C0*/C1*     A5* A4* A3* A2* A1* A0         ;INC BY 1
      +   C0*/C1* CX* A5* A4* A3* A2* A1*/A0         ;INC BY 2

/A7  :=  /LOAD*/A7                                   ;HOLD
      +  /C0* C1* CX*/OP7                            ;LOAD
     :+: /C0*/C1*     A6* A5* A4* A3* A2* A1* A0     ;INC BY 1
      +   C0*/C1* CX* A6* A5* A4* A3* A2* A1*/A0     ;INC BY 2
```

```
SIMULATION

TRACE_ON CLK A7 A6 A5 A4 A3 A2 A1 A0     ;Trace CLK and outputs

SETF OE /C0 C1 CX /OP7 /OP6 /OP5 /OP4 /OP3
CLOCKF CLK                               ;Load all 0s
CHECK /A7 /A6 /A5 /A4 /A3 /A2 /A1 /A0

SETF C0 /C1
CLOCKF CLK                               ;Increment by 2
CHECK /A7 /A6 /A5 /A4 /A3 /A2  A1 /A0    ;to 2

CLOCKF CLK                               ;Increment by 2
CHECK /A7 /A6 /A5 /A4 /A3  A2 /A1 /A0    ;to 4

CLOCKF CLK                               ;Increment by 2
CHECK /A7 /A6 /A5 /A4 /A3  A2  A1 /A0    ;to 6

SETF /C0
CLOCKF CLK                               ;Increment by 1
CHECK /A7 /A6 /A5 /A4 /A3  A2  A1  A0    ;to 7

CLOCKF CLK                               ;Increment by 1
CHECK /A7 /A6 /A5 /A4  A3 /A2 /A1 /A0    ;to 8

SETF C0 C1
CLOCKF CLK                               ;Hold
CLOCKF CLK                               ;Hold
CHECK /A7 /A6 /A5 /A4  A3 /A2 /A1 /A0    ;to 8

TRACE_OFF
```

```
TITLE           AN-100 DSP Condition
PATTERN         DSPCOND
REVISION        1
AUTHOR          Marc Baker
COMPANY         Monolithic Memories
DATE            August 20, 1987

CHIP DSPCOND PAL16C1

CX3  CX2 CX1 CX0 SS  OVF     AUB INR /LLC GND
/CON AC0 AC1 AC2 /CX COMP_CX AC3 NC  NC   VCC

EQUATIONS

CX = /CX3*/CX2*/CX1*/CX0*       CON
   + /CX3*/CX2*/CX1* CX0* SS*   CON
   + /CX3*/CX2* CX1*/CX0* OVF*  CON
   + /CX3*/CX2* CX1* CX0* AUB*  CON
   + /CX3* CX2*/CX1*/CX0*/INR*  CON
   + /CX3* CX2*/CX1* CX0* LLC*  CON
   + /CX3* CX2* CX1*/CX0*/LLC*  CON
   + /CX3* CX2* CX1* CX0*/AC3*/AC2*/AC1*/AC0* CON
   +  CX3*/CX2*/CX1*/CX0*/AC0*  CON
   +  CX3*/CX2*/CX1* CX0* AC0*  CON
   +  CX3*/CX2* CX1*/CX0*/AC1*  CON
   +  CX3*/CX2* CX1* CX0* AC1*  CON
   +  CX3* CX2*/CX1*/CX0*/AC2*  CON
   +  CX3* CX2*/CX1* CX0* AC2*  CON
   +  CX3* CX2* CX1*/CX0*/AC3*  CON
   +  CX3* CX2* CX1* CX0* AC3*  CON

SIMULATION

SETF /CON                       ;SET /CX HIGH
CHECK /CX

SETF CON /CX3 /CX2 /CX1 /CX0    ;CX3-CX0=0 - SET /CX LOW
CHECK CX

SETF CX0 SS                     ;CX3-CX0=1 - SET /CX LOW
CHECK CX

SETF CX1 /CX0 OVF               ;CX3-CX0=2 - SET /CX LOW
CHECK CX

SETF CX0 AUB                    ;CX3-CX0=3 - SET /CX LOW
CHECK CX

SETF CX2 /CX1 /CX0 /INR         ;CX3-CX0=4 - SET /CX LOW
CHECK CX

SETF CX0 LLC                    ;CX3-CX0=5 - SET /CX LOW
CHECK CX
```

```
SETF CX1 /CX0 /LLC                ;CX3-CX0=6 - SET /CX LOW
CHECK CX

SETF CX0 /AC3 /AC2 /AC1 /AC0      ;CX3-CX0=7 - SET /CX LOW
CHECK CX

SETF CX3 /CX2 /CX1 /CX0           ;CX3-CX0=8 - SET /CX LOW
CHECK CX

SETF CX0 AC0                      ;CX3-CX0=9 - SET /CX LOW
CHECK CX

SETF CX1 /CX0                     ;CX3-CX0=10 - SET /CX LOW
CHECK CX

SETF CX0 AC1                      ;CX3-CX0=11 - SET /CX LOW
CHECK CX

SETF CX2 /CX1 /CX0                ;CX3-CX0=12 - SET /CX LOW
CHECK CX

SETF CX0 AC2                      ;CX3-CX0=13 - SET /CX LOW
CHECK CX

SETF CX1 /CX0                     ;CX3-CX0=14 - SET /CX LOW
CHECK CX

SETF CX0 AC3                      ;CX3-CX0=15 - SET /CX LOW
CHECK CX
```

```
TITLE           AN-100 DSP Address Control
PATTERN         ADDCONT
REVISION        1
AUTHOR          Marc Baker
COMPANY         Monolithic Memories
DATE            August 21, 1987

CHIP ADDCONT PAL22V10

CLK OP3 OP2 OP1 /ADE D15 D14 D13 D12 NC NC GND
NC  NC  D11 D10 AC0  AC1 AC2 AC3 D9  D8 NC VCC
GLOBAL

EQUATIONS

AC0 := /OP3*/OP2* OP1*/AC0* ADE                    ;INC
     + /OP3* OP2*/OP1*/AC0* ADE                    ;DEC
     + /OP3* OP2* OP1* D12* ADE                    ;D12
     +  OP3*/OP2*/OP1* D11* ADE                    ;D11
     +  OP3*/OP2* OP1* D8 * ADE                    ;D8
     +  OP3* OP2*/OP1* D14* ADE                    ;MSB EQUATIONS
     +  OP3* OP2*/OP1*/D14*/D13* D12* ADE
     +  OP3* OP2*/OP1*/D14*/D13*/D12*/D11* D10* ADE
     +  OP3* OP2*/OP1*/D14*/D13*/D12*/D11*/D10*/D9*D8* ADE
     +  OP3* OP2* OP1* AC0                          ;HOLD
     +  AC0*/ADE                                   ;HOLD

AC1 := /OP3*/OP2* OP1* AC1*/AC0* ADE               ;INC
     + /OP3*/OP2* OP1*/AC1* AC0* ADE               ;INC
     + /OP3* OP2*/OP1* AC1* AC0* ADE               ;DEC
     + /OP3* OP2*/OP1*/AC1*/AC0* ADE               ;DEC
     + /OP3* OP2* OP1* D13* ADE                    ;D13
     +  OP3*/OP2*/OP1* D12* ADE                    ;D12
     +  OP3*/OP2* OP1* D9 * ADE                    ;D9
     +  OP3* OP2*/OP1* D14* ADE                    ;MSB EQUATIONS
     +  OP3* OP2*/OP1*/D14* D13* ADE
     +  OP3* OP2*/OP1*/D14*/D13*/D12*/D11* D10* ADE
     +  OP3* OP2*/OP1*/D14*/D13*/D12*/D11*/D10* D9* ADE
     +  OP3* OP2* OP1* AC1                          ;HOLD
     +  AC1*/ADE                                   ;HOLD
```

```
AC2 := /OP3*/OP2* OP1*/AC2* AC1* AC0* ADE      ;INC
     + /OP3*/OP2* OP1* AC2*/AC1*      ADE      ;INC
     + /OP3*/OP2* OP1* AC2*     /AC0* ADE      ;INC
     + /OP3* OP2*/OP1*/AC2*/AC1*/AC0* ADE      ;DEC
     + /OP3* OP2*/OP1* AC2* AC1*      ADE      ;DEC
     + /OP3* OP2*/OP1* AC2*      AC0* ADE      ;DEC
     + /OP3* OP2* OP1* D14* ADE                 ;D14
     +  OP3*/OP2*/OP1* D13* ADE                 ;D13
     +  OP3*/OP2* OP1* D10* ADE                 ;D10
     +  OP3* OP2*/OP1* D14* ADE                 ;MSB EQUATIONS
     +  OP3* OP2*/OP1* D13* ADE
     +  OP3* OP2*/OP1* D12* ADE
     +  OP3* OP2*/OP1* D11* ADE
     +  OP3* OP2* OP1* AC2                      ;HOLD
     +  AC2*/ADE                                ;HOLD

AC3 := /OP3*/OP2* OP1*/AC3* AC2* AC1* AC0* ADE  ;INC
     + /OP3*/OP2* OP1* AC3*/AC2*           ADE  ;INC
     + /OP3*/OP2* OP1* AC3*     /AC1*      ADE  ;INC
     + /OP3*/OP2* OP1* AC3*          /AC0* ADE  ;INC
     + /OP3* OP2*/OP1*/AC3*/AC2*/AC1*/AC0* ADE  ;DEC
     + /OP3* OP2*/OP1*/AC3* AC2*           ADE  ;DEC
     + /OP3* OP2*/OP1*/AC3*      AC1*      ADE  ;DEC
     + /OP3* OP2*/OP1*/AC3*           AC0* ADE  ;DEC
     + /OP3* OP2* OP1* D15* ADE                 ;D15
     +  OP3*/OP2*/OP1* D14* ADE                 ;D14
     +  OP3*/OP2* OP1* D11* ADE                 ;D11
     +  OP3* OP2* OP1* AC3                      ;HOLD
     +  AC3*/ADE                                ;HOLD

SIMULATION

TRACE_ON CLK AC3 AC2 AC1 AC0

SETF ADE /OP3 /OP2 /OP1
CLOCKF CLK                          ;CLEAR TO 0
CHECK /AC3 /AC2 /AC1 /AC0

SETF OP1
CLOCKF CLK                          ;INCREMENT TO 1
CHECK /AC3 /AC2 /AC1 AC0

CLOCKF CLK                          ;INCREMENT TO 2
CHECK /AC3 /AC2 AC1 /AC0

CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK                          ;INCREMENT TO 8
CHECK AC3 /AC2 /AC1 /AC0
```

```
SETF OP2 /OP1
CLOCKF CLK                          ;DECREMENT TO 7
CHECK /AC3 AC2 AC1 AC0

CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK                          ;DECREMENT TO 3
CHECK /AC3 /AC2 AC1 AC0

SETF OP1 /D15 D14 /D13 D12 /D11
CLOCKF CLK                          ;SET TO D15-D12
CHECK /AC3 AC2 /AC1 AC0

SETF OP3 /OP2 /OP1
CLOCKF CLK                          ;SET TO D14-D11
CHECK AC3 /AC2 AC1 /AC0

SETF OP1 D10 /D9 D8
CLOCKF CLK                          ;SET TO D11-D8
CHECK /AC3 AC2 /AC1 AC0

SETF OP2 /OP1
CLOCKF CLK                          ;CHECK MSB
CHECK /AC3 AC2 AC1 AC0

SETF /D14
CLOCKF CLK                          ;CHECK MSB
CHECK /AC3 AC2 /AC1 AC0

SETF OP1
CLOCKF CLK                          ;HOLD
CHECK /AC3 AC2 /AC1 AC0

SETF /OP3 /OP2 /OP1 /ADE            ;HOLD
CLOCKF CLK
CHECK /AC3 AC2 /AC1 AC0

TRACE_OFF
```

Bus Interface

Bus interface plays an important role in computer architecture design. As systems become more complicated, the bus requirement becomes more critical in terms of speed, power, accessibility and feasibility. There are many different kinds of buses in the electronics industry. Each of the buses is specially designed to fit into a special architecture for various purposes. There are many stringent rules and regulations to follow depending on the application to design a bus architecture or bus interface. The application depends on timing (synchronous or asynchronous), type (military or commercial), architecture (open or closed), data structure (serial or parallel), computer type (micro, mini or others) or a mix of the above.

There have been nearly one hundred buses since the invention of the computer. A great deal of them have faded away because the application no longer exists, or have been replaced by new technology.

The following are some current major buses:

- DEC Unibus
- Future Bus
- IEEE 488 Bus
- Multibus II
- Nu Bus
- PC Bus
- VME Bus
- Q Bus

Disregarding the complexity and different varieties of buses, there are four basic sections (Figure 1) in the bus structure:

- Address Section
- Data Section
- Control Section
- Utility Section

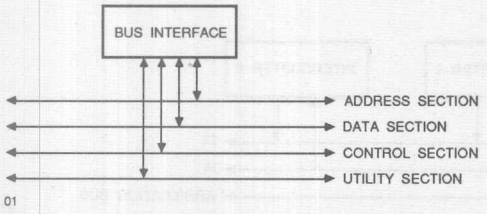


Figure 1. Four Sections of the Bus Interface

Address Section

The Address Section consists of the address lines and control portion. The address lines are used by the processor to indicate to memory and other peripherals the location with which it wants to communicate. The address width varies from bus to bus. For example, VME, Nu and Multibus II are 32 bits wide. DEC Unibus, PC Bus and Q Bus are 18, 20 and 22 bits wide, respectively. The width of the address line has gradually expanded with each new bus announcement. As the width expands, the address decoding becomes more complex. One solution to this complex address decoding problem is to use PAL devices such as the PAL16L8/20L8/20L10. The advantage of using these PAL devices is to utilize their wide input range (up to 16/20) to address up to 64/1M addresses by a single PAL device.

Data Section

The Data Section consists of the data lines and control portion. The data lines carry instruction and data between the processor and the peripherals, including the memory. The control portion provides support to the data lines. In most cases, data can be addressed as byte, word or double-word. For instance, PC Bus, DEC Unibus, and Nu Bus can be addressed as 8, 16 and 32-bits wide, respectively. VME and Multibus II can both be addressed as 8, 16, 24 or 32-bits wide. The external bus data communication requires handshaking. As long as the protocol is well defined, it can be implemented easily by the PAL16R8/20R8/20X10. If a change is needed after the circuit is designed, we can simply modify the PAL device file. In other words, by using PAL devices for data communication, flexibility can be achieved with minimum hardware change.

One of the major problems when dealing with data transfer is synchronization. Synchronization is when events occur concurrently with a regular or predictable timing relationship, typically based on a system clock. The data transfer control functions for this type of synchronous bus can be implemented by PAL devices such as the 20R8. In asynchronous buses, the data lines communicate through handshaking instead of the data clock. This can be tedious when using discrete logic. The asynchronous devices PAL16RA8/20RA10 can be very useful for implementing such designs.

Another problem is when both ends of the data lines have different data rates. In this case we need FIFO memories. FIFO memories match the data rates and serve as temporary buffers between source and destination.

Control Section

The Control Section coordinates the operation of all bus interface circuitry. This is the heart of the bus interface design. The major architecture differences can be observed here. Generally, any device that is capable of controlling bus operation is called a "bus master". Devices that operate on the bus but cannot control it, i.e. the memory system, I/O ports, etc. are called "slaves". Most of the buses are designed to allow more than one bus master or a variety of intelligent devices, i.e. a second CPU, disk controller or network controller, on the bus.

Functionally, the Control Section can be split into three subsections:

- Interrupt Subsection
- DMA Subsection
- Arbiter Subsection

Interrupt Subsection

Interrupts are important to all bus systems. These signals are designed to allow infrequent or important peripherals to get the attention of the bus master. A maskable interrupt can be ignored by the bus master, while a non-maskable interrupt can not. Vectored interrupt inputs provide priority to the interrupt controller in case more than one interrupt request should occur simultaneously. The number and type of interrupt inputs vary widely among buses.

To determine which device caused the interrupt in order to execute the correct interrupt-handling routine, two methods can be used:

1. Software Method—In this method, a polling program checks, one by one, each of the devices connected to the interrupt line, to determine which one is interrupting. When an interrupting device is found, the routine then branches to the appropriate interrupt handler. This scheme is considerably slow due to the software overhead involved.

2. Hardware Method—This method is more efficient than the software scheme. The interrupted peripheral will provide an interrupt vector which forms the branching address to the interrupt-handling routine at the time it occurs. This is called automated interrupt-vectoring (Figure 2). After the Interrupt Request (IR) and the vector are issued by a peripheral, the Interrupt Handler arbitrates, grants and serves that interrupt. The VME and DEC Unibus use this interrupt technique. The required branching address is loaded in the address register. Ordinary registered PAL devices such as the 20R8/20X10 together with the 20L8/20L10 are suitable for this technique. A design example of the interrupt controller for the DEC Unibus is shown on page 2-328. This example demonstrates a near 4 to 1 chip count saving by using a PAL device design. Obviously, the real estate and debugging time shrink and the reliability increases.

In the event that several interrupts occur simultaneously, a priority scheme is used to decide which I/O device should be granted control of the bus. The most popular one is the Fixed Priority Scheme. This assigns a fixed priority number to each device. This number will be there until the next configuration change. The other scheme is more complex and is rarely used, called the Programmable Priority Scheme. In this scheme, the priority can be changed dynamically by the software program. This scheme may be useful in applications such as strategic military or dynamic financial marketing where the strategy changes with various battles or investment environments; but, the software design may be very complex. However, both schemes can be accomplished by using sequencer devices such as the PMS14R21. This powerful device can serve as a local distributed processor within the Bus Interface Unit.

Direct Memory Access (DMA) Subsection

The Direct Memory Access concept exists because the ordinary memory transferring speed often isn't fast enough. The transfer speed of a microprocessor is limited by the software overhead involved in transferring successive words: several instructions must be executed in order to transfer a word or a chain of words. In such cases where the ordinary memory transferring speeds may not be sufficient, as in a floppy disk or a frame buffer, DMA is required. The DMA Controller implements the transfer algorithm in hardware and automates the word or block transfers directly between the memory and the I/O device.

Two techniques are used:

1. The processor is halted or suspended by the DMA Controller (DMAC). The DMA dominates the bus until the end of the transfer.

2. The DMAC steals a memory cycle from the processor and inserts a data transfer between the memory and the I/O device. This is called cycle-stealing DMA.

These methods can also be used in a combined fashion.

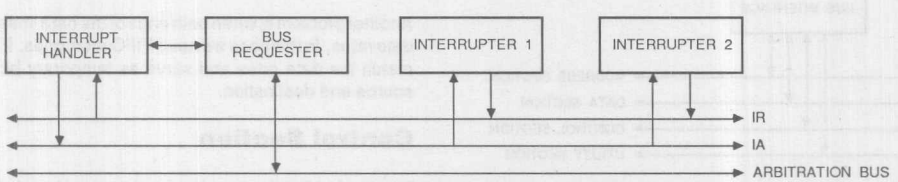


Figure 2. Bus Interrupt Example

DMAC design can involve complex handshake sequences between the processor and an I/O device. It contributes significantly to the system's cost and complexity and thus it is not normally used in small systems. However, with the availability of PAL devices, DMA applications are becoming more popular. The handshake sequence and complex discrete logic design can be replaced by asynchronous PAL devices such as the PAL16RA8/20RA10 and programmable sequencers such as the PMS14R21/A. This can significantly decrease the cost, real estate and debugging time.

Arbiter Subsection

Bus Arbiter designs are becoming more critical and complex as microprocessor costs decrease. This is because it is becoming more cost-effective to design systems with multiple processors sharing global resources. In the bus interface, the arbiter acts as a judge to decide who should own the proprietorship of the bus and for how long. The arbiter decides which Requester should be granted control of the Bus when several requests happen simultaneously.

The function of the arbiter consists of:

1. Granting the use of the bus to one of the bus masters.
2. Scheduling requests from multiple bus masters for optimum bus use.
3. Preventing simultaneous use of the bus by two bus masters.

In other words, a successful arbiter design can prevent bus hangup and increase bus bandwidth.

Four major types of arbitration are:

1. Prioritized Arbiter—This priority scheme provide a pre-assigned number to each bus master to arbitrate the execution order.
2. Round-robin Arbiter—This scheme assigns the bus master on a rotating priority basis.
3. Single-Level Arbiter—This scheme provides only single level arbitration. However, it relies on a bus grant daisy-chain to arbitrate the request.
4. Fairness Arbiter—This scheme prohibits the winning module from being serviced again until all pending requests have been serviced.

An example of using the PAL22V10 to perform the Prioritized Arbiter is described on the application note on page 2-347. The arbitration process and PAL device equations are also provided.

An example of an arbiter used for resolving a bus contention between two bus masters fighting for control is shown in Figure 3.

When both master 1 (M_1) and master 2 (M_2) issue a Bus Request (BR), the Arbiter will grant the Bus Grant (BG) to the one with higher priority. Assuming it is M_2, M_2 will then drive Bus Busy (B Busy) active and release the BR signal. After M_2 finishes its operation, the Arbiter detects that the BR signal is still low and transfers the bus control to M_1.

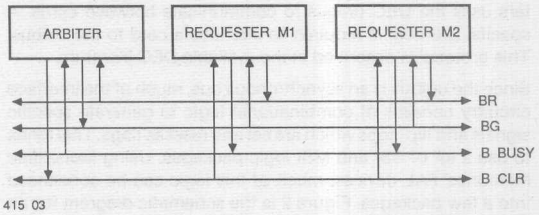


Figure 3. Arbitration Example: Bus Contention

Utility Section

The Utility section provides support to each of the previous sections. It links the entire bus interface between modules. The Utility Section consists of the System Clock Distribution, Power Monitor and System Diagnostics. The Power Monitor detects power failures and signals the system to start an orderly shutdown. When power is then reapplied to the system, it ensures that each module is initialized accordingly. For example, the power failure and power up sequences for the VME bus are 4 ms and 200 ms, respectively. This can help the processor to react to these external interrupts according to the system specification. A system without diagnostics is an incomplete system. System Diagnostics can be implemented by a Programmable Sequencer such as the PMS14R21/A (page 3-123).

Future System Expansion

With the rapid rate of improvement of technology, the useful life of a new system is increasingly getting shorter. Therefore, the future system expansion consideration can be a major design feature to system designers as well as a selection guideline to a buyer. The key to system expansion in a general computer system is the bus through which the CPU communicates with the present and future system components. Design flexibility, high throughput, multiprocessor support, and low power consumption soon become critical design considerations for bus interface techniques. PLDs can be the solution for bus interface, not only because of their flexibility in terms of possible hardware modification, but also because of their real estate saving, which in turn lowers the manufacturing cost and power consumption.

Unibus Interrupt Controller

Functional Description

One of the more established computer families is the Digital Equipment Corporation's PDP-11 series. This family of computers uses the DEC unibus to communicate between cards. A specific protocol is required to interface a card to the unibus. This protocol is described in the available DEC literature.

Since the unibus is an asynchronous bus, much of the interface circuitry consists of combinational logic to generate specific signals and flip-flops which are set and reset as flags. This tends to use a lot of SSI and MSI logic packages. Using Monolithic Memories' PAL devices, much of this logic can be condensed into a few packages. Figure 2 is the schematic diagram for an interrupt controller to be used on the unibus.

Many cards communicate over the bus by taking control of the unibus with an interrupt request, and then do whatever they require before releasing control. As can be seen, this interrupt controller takes six special interface ICs, (380 and 8881 bus drivers and receivers) eight MSI, SSI ICs, (7400, 7402 and 7474s) along with some transistors and discrete parts. This parts count can be considerably reduced by using a PAL20RA10 and a PAL20L10.

Figure 1 shows how the circuit with the PAL devices would look. The two PAL devices allow almost all of the 7400, 7402 and 7474 packages to be removed. (Almost a four-to-one saving in chip count.) In addition, the preload pin (PRLD) on the 20RA10 allows the flip-flops to be easily set to a known state on power up, or when reinitializing. So the PAL devices reduce the logic package count from eight chips to three.

In the schematic shown, there are three VLSI devices, three MSIs and two SSIs. Using a PAL20RA10, it is possible to replace three MSIs and one SSI device, thereby reducing the chip count by a factor of two. The ICs inside the enclosed loop were replaced.

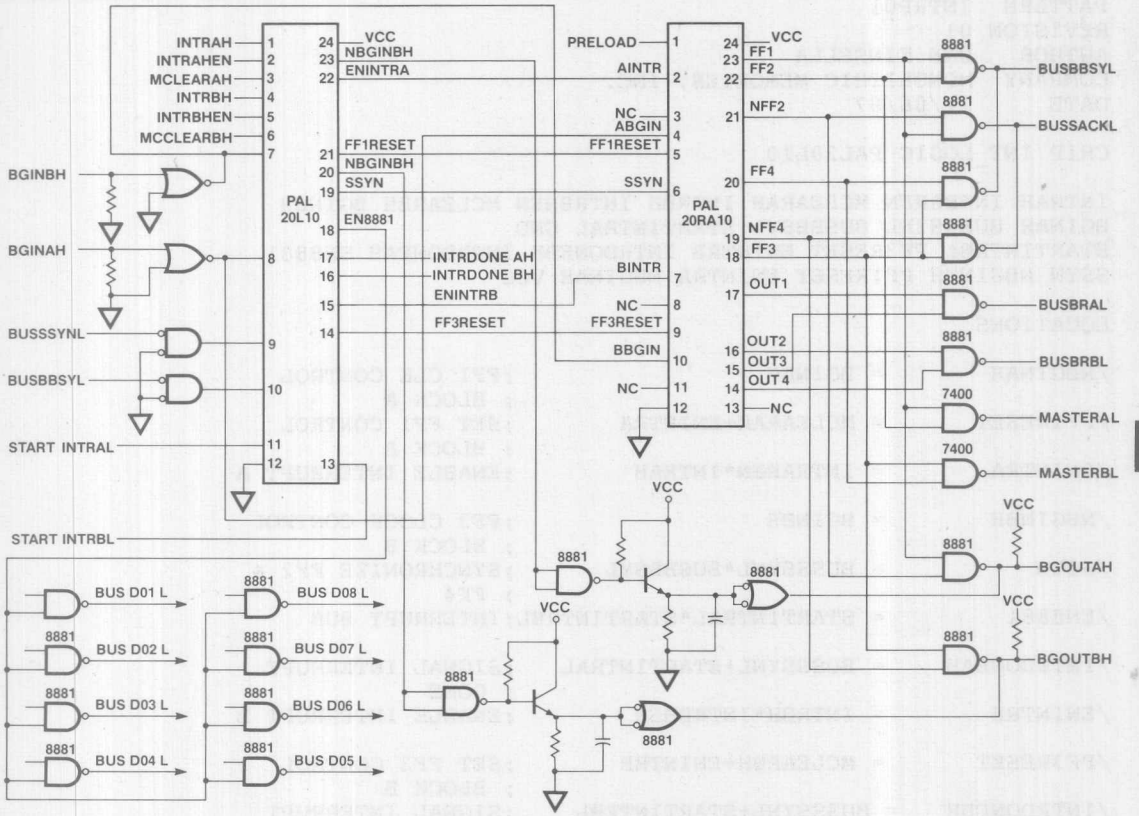

INTRAH
INTRAHEN
MCLEARAH
INTRBH
INTRBHEN
MCCLEARBH
VCC
NBGINBH
ENINTRA
FF1RESET
NBGINBH
SSYN
EN8881
PAL
20L10
INTRDONE AH
INTRDONE BH
ENINTRB
FF3RESET
BGINBH
BGINAH
BUSSSYNL
BUSBBSYL
START INTRAL
START INTRBL
PRELOAD
AINTR
NC
ABGIN
FF1RESET
SSYN
PAL
20RA10
BINTR
NC
FF3RESET
BBGIN
NC
VCC
FF1
FF2
NFF2
FF4
NFF4
FF3
OUT1
OUT2
OUT3
OUT4
NC
8881
BUSBBSYL
BUSSACKL
BUSBRAL
BUSBRBL
7400
MASTERAL
MASTERBL
BGOUTAH
BGOUTBH
BUS D01 L
BUS D02 L
BUS D03 L
BUS D04 L
BUS D08 L
BUS D07 L
BUS D06 L
BUS D05 L

Figure 1.

```
TITLE     INTERRUPT LOGIC
PATTERN   INTRP01
REVISION  01
AUTHOR    DAN KINSELLA
COMPANY   MONOLITHIC MEMORIES, INC.
DATE      11/06/87

CHIP INT_LOGIC PAL20L10

INTRAH INTRAHEN MCLEARAH INTRBH INTRBHEN MCLEARBH BGINBH
BGINAH BUSSSYNL BUSBBSYL STARTINTRAL GND
STARTINTRBL FF3RESET ENINTRB INTRDONEBH INTRDONEAH EN8881
SSYN NBGINBH FF1RESET ENINTRA NBGINAH VCC

EQUATIONS

/NBGINAH        = BGINAH                    ;FF1 CLK CONTROL
                                            ; BLOCK A
/FF1RESET       = MCLEARAH+ENINTRA          ;SET FF1 CONTROL
                                            ; BLOCK A
/ENINTRA        = INTRAHEN*INTRAH           ;ENABLE INTERRUPT A

/NBGINBH        = BGINBH                    ;FF3 CLOCK CONTROL
                                            ; BLOCK B
/SSYN           = BUSSSYNL*BUSBBSYL         ;SYNCHRONIZE FF2 &
                                            ; FF4
/EN8881         = STARTINTRAL*STARTINTRBL;INTERRUPT BUS

/INTRDONEAH     = BUSSSYNL+STARTINTRAL      ;SIGNAL INTERRUPT
                                            ; DONE
/ENINTRB        = INTRBH*INTRBHEN           ;ENABLE INTERRUPT B

/FF3RESET       = MCLEARBH+ENINTRB          ;SET FF3 CONTROL
                                            ; BLOCK B
/INTRDONEBH    = BUSSSYNL+STARTINTRBL       ;SIGNAL INTERRUPT
                                            ; DONE

;DESCRIPTION

;COMBINATORIAL LOGIC FOR PAL20RA10 INTERRUPT CONTROLLER
;(1ST PART OF THE TWO PAL SOLUTION:  PAL20L10 & PAL20RA10)

; SIMULATION NOT INCLUDED
```

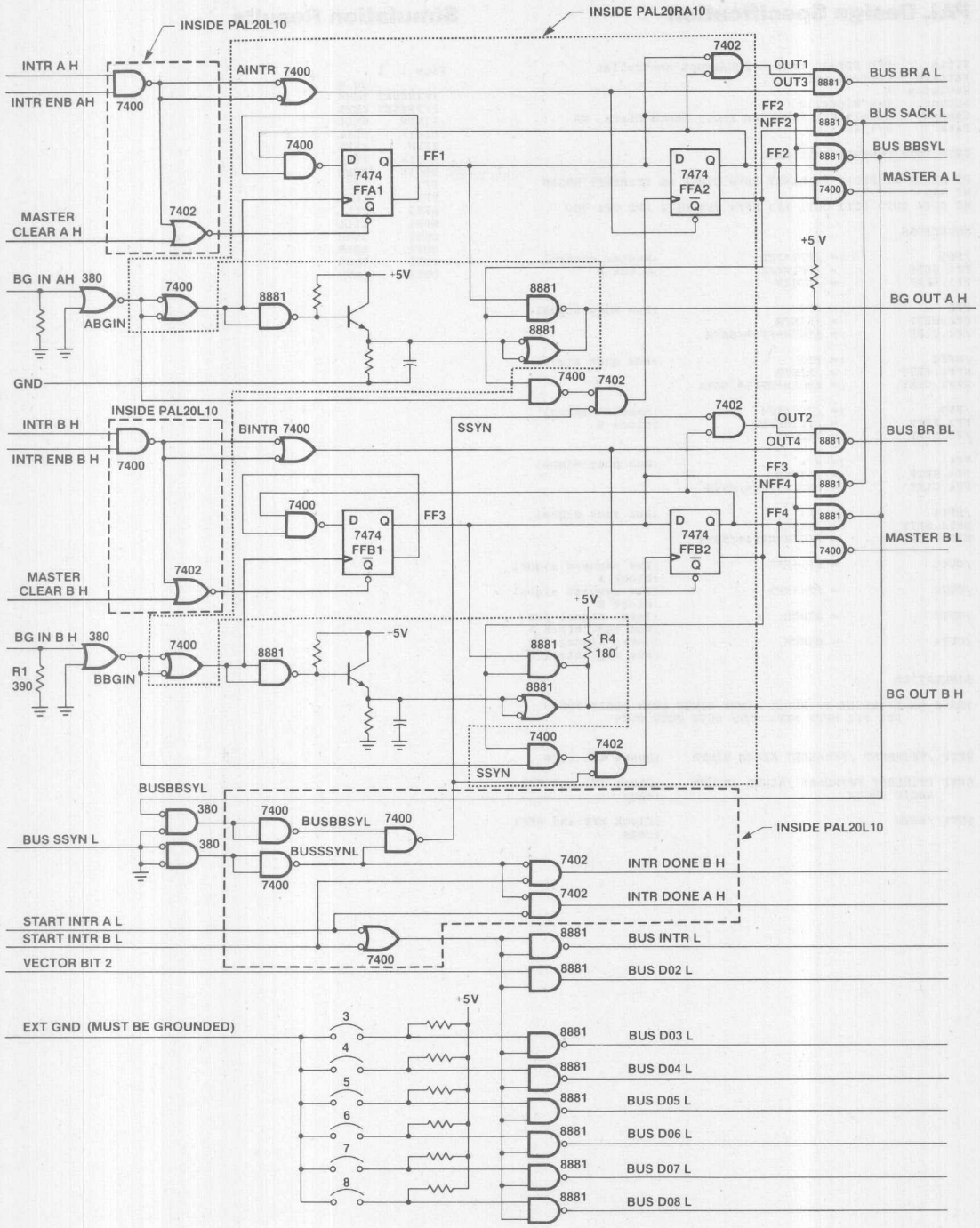

INSIDE PAL20L10
INSIDE PAL20RA10
INTR A H
INTR ENB AH
7400
AINTR 7400
7402
OUT1
OUT3
8881
BUS BR A L
FF2
NFF2
BUS SACK L
FF1
D Q
7474
FFA1
Q
D Q
7474
FFA2
Q
FF2
BUS BBSYL
MASTER A L
7400
MASTER
CLEAR A H
+5 V
BG IN AH
380
ABGIN
+5V
BG OUT A H
GND
INSIDE PAL20L10
INTR B H
INTR ENB B H
BINTR 7400
SSYN
OUT2
OUT4
BUS BR BL
FF3
NFF4
FF4
D Q
7474
FFB1
Q
D Q
7474
FFB2
Q
MASTER B L
MASTER
CLEAR B H
BG IN B H
R1
390
BBGIN
R4
180
BG OUT B H
BUSBBSYL
BUS SSYN L
BUSSSYNL
INSIDE PAL20L10
INTR DONE B H
INTR DONE A H
START INTR A L
START INTR B L
VECTOR BIT 2
BUS INTR L
BUS D02 L
EXT GND (MUST BE GROUNDED)
BUS D03 L
BUS D04 L
BUS D05 L
BUS D06 L
BUS D07 L
BUS D08 L

Figure 2.

PAL Design Specification

```
Title     DEC PDP-11 unibus interrupt controller
Pattern   Control.pds
Revision  A
Author    Dan Kinsella
Company   Monolithic Memories Inc., Santa Clara, CA
Date      3/1/85

CHIP INTR_CONTROL PAL20RA10

PL AINTR NC ABGIN FF1RESET SSYN BINTR NC FF3RESET BBGIN
NC GND
NC OUT4 OUT3 OUT2 OUT1 FF3 NFF4 FF4 NFF2 FF2 FF1 VCC

EQUATIONS

/FF1              := /FF1*FF2              ;Master control
FF1.SETF           = /FF1RESET             ;block A
FF1.CLKF           = /ABGIN

FF2               := FF1                   ;Bus Busy Signal
FF2.SETF           = /AINTR
FF2.CLKF           = ABGIN*FF2*/SSYN

/NFF2             := FF1                   ;Bus sack signal
NFF2.SETF          = /AINTR
NFF2.CLKF          = ABGIN*NFF2*/SSYN

/FF3              := /FF3*FF4              ;Master control
FF3.SETF           = /FF3RESET             ;block B
FF3.CLKF           = /BBGIN

FF4               := FF4                   ;Bus busy signal
FF4.SETF           = /BINTR
FF4.CLKF           = BBGIN*FF4*/SSYN

/NFF4             := FF3                   ;Bus sack signal
NFF4.SETF          = /BINTR
NFF4.CLKF          = BBGIN*NFF4*/SSYN

/OUT1              = FF1+FF2               ;Bus request signal
                                           ;block A
/OUT2              = FF4+FF3               ;Bus request signal
                                           ;block B
/OUT3              = AINTR                 ;Intr. signal for
                                           ;bus req. block A
/OUT4              = BINTR                 ;Intr. signal for
                                           ;bus req. block B

SIMULATION

TRACE_ON FF1RESET FF3RESET AINTR BINTR SSYN ABGIN BBGIN
         FF1 FF3 NFF2 NFF4 OUT1 OUT2 OUT3 OUT4


SETF /FF1RESET /FF3RESET AINTR BINTR     ;Reset all regs

SETF FF1RESET FF3RESET /AINTR /BINTR     ;Clock FF1 and FF3
     ABGIN BBGIN                         ;regs

SETF /SSYN                               ;Clock NFF and NFF3
                                         ;regs
```

Simulation Results

```
Page :  1
                gg g
 FF1RESET       LHHH
 FF3RESET       LHHH
 AINTR          HLLL
 BINTR          HLLL
 SSYN           XXXL
 ABGIN          XHHH
 BBGIN          XHHH
 FF1            LLLL
 FF3            LLLL
 NFF2           XLLL
 NFF4           XLLL
 OUT1           XHHH
 OUT2           XHHH
 OUT3           LHHH
 OUT4           LHHH
```

A Multibus Arbiter Design for 10 MHz Processors

This application note describes the implementation of a bus arbiter using a PAL16R4 to interface the 10 MHz 80186 to the MULTIBUS multi-master environment. The PAL equations for bus exchange were developed based on functional and timing requirements specified in the Intel MULTIBUS Specification. The interface shown addresses the bus master case in regards to bus exchange only. No attempt has been made to include the requirements for bus slave operation, interrupt handling or byte operations. This note assumes a knowledge of MULTIBUS operations and signal names.

Overview

The system environment is assumed to be a collection of loosely coupled independent processors which communicate through a common memory on the system bus. Each processor in this system possesses its own local resources.

A block diagram showing the interface of one such processor, an 80186, to the system bus, MULTIBUS, is shown in Figure 1. A block of the 80186's address space is allocated to system memory. Access into this portion of the address map generates a request for the system bus via a programmable memory chip-select pin. Once a request is generated by the CPU, the PAL16R4 arbiter and the 8288 bus controller combine to perform a bus exchange and transfer cycle.

In this design the arbiter maintains control of the system bus and therefore must be forced off either by a higher priority master ($\overline{BPRN}$ goes HIGH) or a Common Bus Request ($\overline{CBRQ}$).

Functional Requirements

On the MULTIBUS, the bus exchange process begins when the Bus Request ($\overline{BREQ}$) goes LOW (parallel priority resolution scheme), or when $\overline{BPRO}$ goes HIGH to succeeding masters (serial priority resolution scheme). If the requesting master is granted priority ($\overline{BPRN}$ enabled) and the bus is not under control of another master ($\overline{BUSY}$ disabled), then the master may take control of the bus by forcing $\overline{BUSY}$ LOW.

An optional request mechanism, Common Bus Request, used in conjunction with either parallel or serial priority has been implemented in this design.

It reduces bus acquisition overhead by allowing bus retention across transfer cycles, unless a request is pending ($\overline{CBRQ}$ is LOW). For example, in parallel priority the requesting master asserts both $\overline{BREQ}$ and $\overline{CBRQ}$ to force the current master off the bus. Once acquisition is complete $\overline{CBRQ}$ is released for use by another master.

Timing Considerations

Bus exchange on MULTIBUS is a synchronous process. $\overline{BREQ}$, $\overline{BPRO}$, $\overline{BUSY}$ and $\overline{CBRQ}$ are all synchronized with the trailing edge (HIGH-to-LOW transition) of BCLK. BCLK is a MULTIBUS signal having a duty cycle of approximately 50% and a maximum frequency of 10 MHz. There is no requirement for synchronization between BCLK and any other clock in a MULTIBUS system. Synchronizer circuits must be used on signals that cross clock boundaries, because it is possible (as in this design) for the processor clock to be asynchronous to BCLK.

Bus priority on MULTIBUS resolution takes place in one BCLK cycle. If the serial priority scheme is implemented, this requirement will place an upper limit on the number of masters allowed for a given BCLK rate. The parallel priority scheme allows the relationship between BCLK and the number of masters to be independent. Whichever is chosen, the MULTIBUS specified minimum setup time of 22 ns (resolution to clock), plus the desire to keep the $\overline{BPRN}$ to $\overline{BPRO}$ propagation delay to a minimum, points to the use of an "A" speed PAL device (Figure 2).

The MULTIBUS bus exchange timing requires a BUSY setup time before clock of 25 ns (see Figure 2). Release by one master and acquisition by another takes place on two successive HIGH to LOW transitions of BCLK.

PAL Design Description

The PAL design specification with supporting timing diagrams are shown in Figures 3 and 4. The following remarks are in addition to the comments found on the PAL design specification.

The term "BRQP*AEN" found in the equation for "$\overline{BREQ}$" is required to keep the arbiter from recapturing the bus when it is the releasing master. This can occur when a $\overline{CBRQ}$ forced the surrender, and the processor cycle which follows the just completed cycle accesses the system memory.

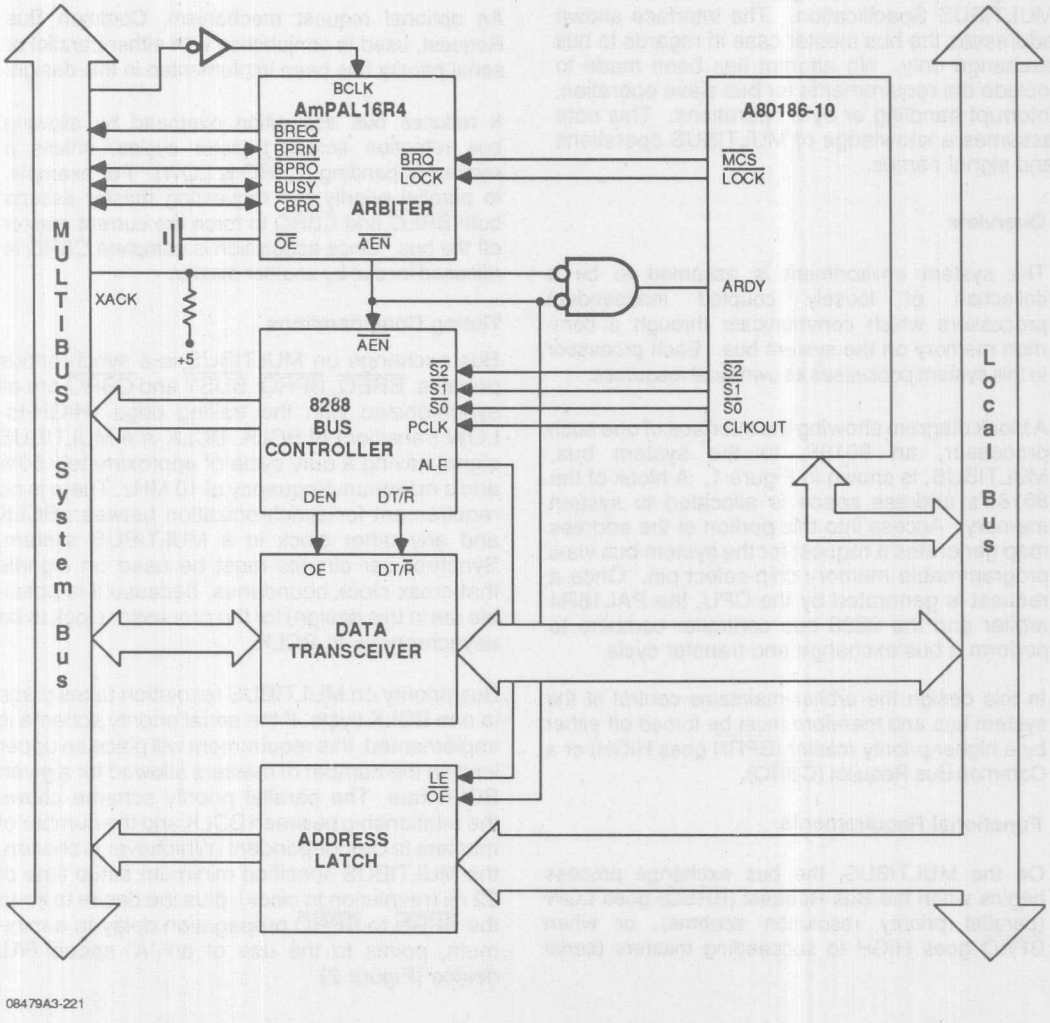


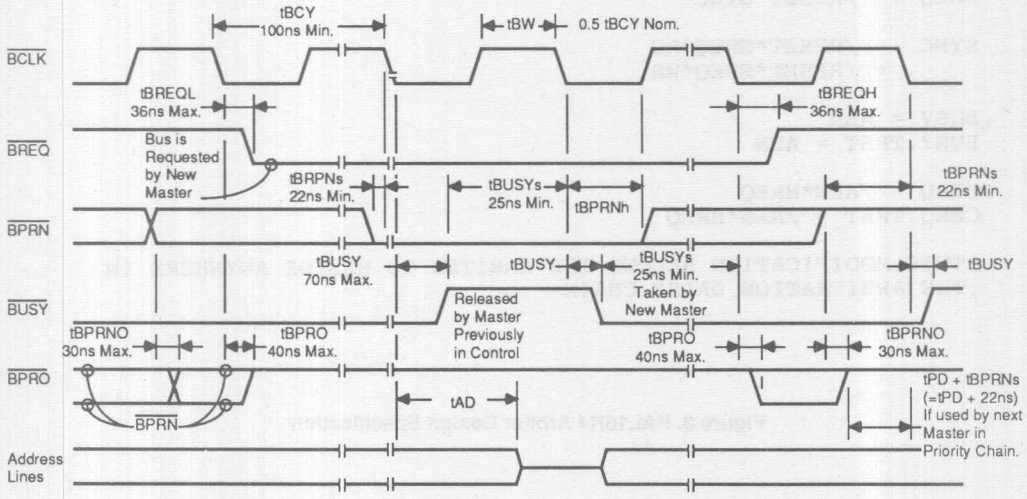
Figure 1. Single Multimaster Bus Interface

Using $\overline{BREQ}$ to disable $\overline{BPRO}$ results in a 40 ns delay worst case from clock (tCO + tPD). Taking into account the 22 ns setup (see Fig. 2) and clock skew (3 ns max) leaves 35 ns of cycle (10 MHz BCLK), if the serial priority resolution scheme is being employed. Lower priority masters using this design would exhibit a 25 ns worst case delay $\overline{BPRN}$ to $\overline{BPRO}$. Use an external gate to implement the equation "$\overline{BPRO} = \overline{BPRN} * \overline{BREQ}$" if the serial priority is required when using more than two masters.

Note the inversion of BLCK for use by the PAL device. PAL device registers load data on the LOW to HIGH clock transition of pin 1.

MULTIBUS address setup before Command (50 ns min.) is determined by the 8288 bus controller. The 8288 specification guarantees 115 ns minimum between $\overline{AEN}$ going LOW and Commands going active.

MULTIBUS address hold after Command (50 ns min.) is satisfied by using the ALE signal from the 8288 instead of the 80186. The 8288 Command lines go inactive on the HIGH to LOW transition of CLKOUT at the beginning of T4 (30 ns max delay). The 8288 ALE signal goes active on the HIGH to LOW transition of CLKOUT at the beginning of T1 (1/2 clock cycle later than ALE from the 80186). The minimum hold time is therefore 100 ns - 35 ns = 65 ns.



08479A3-222

Figure 2. Bus Exchange AC Timing

```
TITLE     A MULTIBUS ARBITER
PATTERN   AMP3-318
REVISION  005
AUTHOR    JOE ENGINEER
COMPANY   ADVANCED MICRO DEVICES
DATE      11/15/87

CHIP MULTIB PAL16R4

BCLK /RD /WR /SREQ /RESET /BPRN NC NC NC GND
/E /CBRQ /BUSY /SYNC /HREQ /AEN /OEN /BREQ /BPRO VCC

EQUATIONS

BPRO = /HREQ*BPRN

BREQ = HREQ+AEN

OEN := /RESET*SREQ*AEN

AEN := /RESET*AEN*HREQ*RD          ;HOLD BUS UNTIL CURRENT
     + /RESET*AEN*HREQ*WR          ;CYCLE IS COMPLETE
     + /RESET*HREQ*BPRN*/BUSY      ;INITIAL BUS ACQUISITION
     + /RESET*AEN*BPRN*/CBRQ       ;HOLD BUS IF NO OTHER REQUEST

HREQ := /RESET*SYNC

SYNC := /RESET*SREQ*RD
      + /RESET*SREQ*WR

BUSY = AEN
BUSY.TRST = AEN

CBRQ = /AEN*HREQ
CBRQ.TRST = /AEN*HREQ

;THIS MODIFICATION ALLOWS THE ARBITER TO RESIDE ANYWHERE IN
;THE ARBITRATION DAISY CHAIN
```

Figure 3. PAL16R4 Arbiter Design Specification

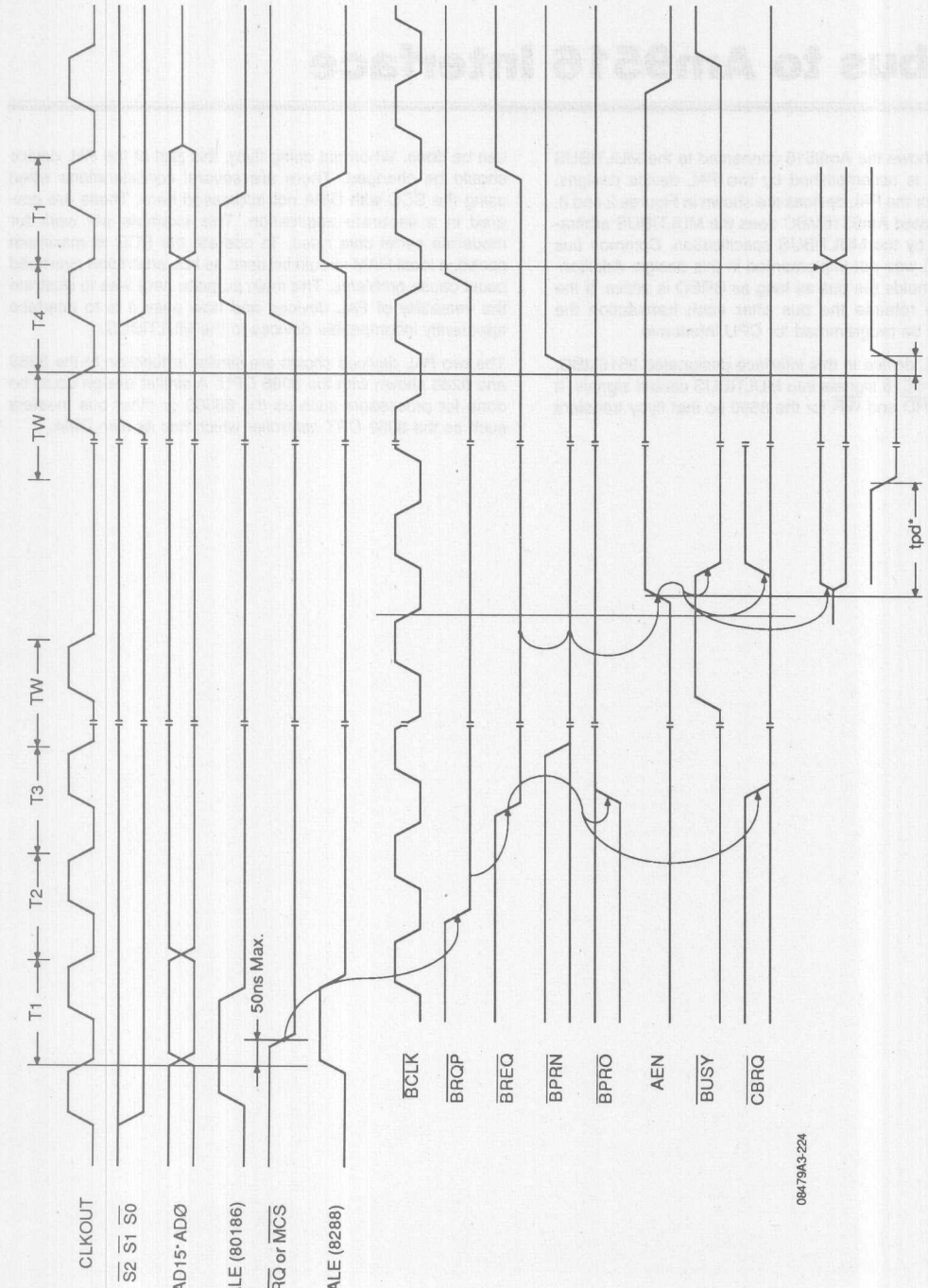


Figure 4. Bus Transfer Timing Diagram

Multibus to Am9516 Interface

This interface shows the Am9516 connected to the MULTIBUS (Figure 1). This is accomplished by two PAL device designs. The equations for the PAL devices are shown in Figures 2 and 3. The first designated Am9516MBC does the MULTIBUS arbitration as defined by the MULTIBUS specification. Common bus request ($\overline{CBRQ}$) was not implemented in this design. Additionally this design holds the bus as long as BREQ is active. If the user wishes to release the bus after each transaction the Am9516 should be programmed for CPU interleave.

The second PAL device in this interface designated 9516MBC, converts the Am9516 signals into MULTIBUS control signals. It also generates $\overline{RD}$ and $\overline{WR}$ for the 8530 so that flyby transfers can be done. When not doing flyby, this part of the PAL device should be changed. There are several considerations when using the SCC with DMA not addressed here. These are covered in a separate application. This example will work for moderate serial data rates. To operate the SCC at maximum speed, a local RAM should be used as bus arbitration overhead could cause problems. The main purpose here was to illustrate the versatility of PAL devices and how easy it is to interface apparently incompatible devices to the MULTIBUS.

The two PAL devices shown are similar in function to the 8289 and 8288 shown with the 8086 CPU. A similar design could be done for processors such as the 68000 or other bus masters such as the 8052 CRT controller which has its own DMA.

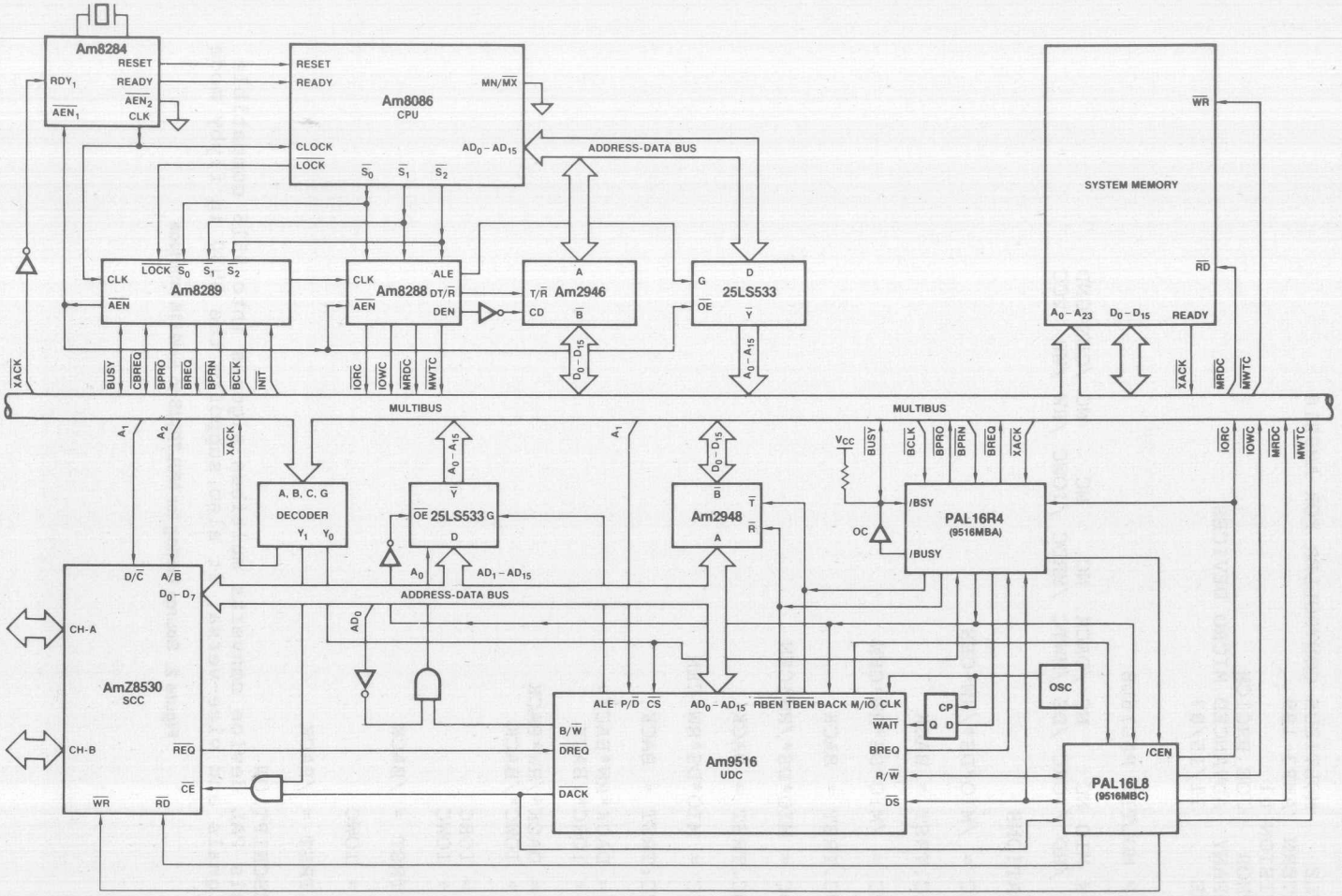


Figure 1. Am9516 to 8086 in a MULTIBUS® Environment

02188A-128

```
TITLE    MULTIBUS CONTROLLER FOR AM9516
PATTERN  AMP3-189
REVISION 01
AUTHOR   JOE BRCICH
COMPANY  ADVANCED MICRO DEVICES
DATE     11/15/87


CHIP MULTI1 PAL16L8

BACK MIO NC    NC /DACK  NC    NC    NC /CEN GND
NC  /RD /IORC /DS /MWTC /MRDC /IOWC /RW /WR  VCC

EQUATIONS

IORC = /MIO*DS*/RW*CEN

IORC.TRST = BACK

IOWC = /MIO*DS*RW*CEN

IOWC.TRST = BACK

MRDC = MIO*DS*/RW*CEN

MRDC.TRST = BACK

MWTC = MIO*DS*RW*CEN

MWTC.TRST = BACK

RD = DACK*RW*BACK
   + IORC*/BACK

WR = DACK*/RW*BACK
   + IOWC*/BACK

DS = IORC
   + IOWC

DS.TRST = /BACK

RW = IOWC

RW.TRST = /BACK

;DESCRIPTION
;This PAL device converts multibus signals into 9516 compatible
;signals and vice-versa. It also supports the 8530 in flyby mode
```

Figure 2. Source Listing for MULTIBUS to Am9516 Interface

```
TITLE     MULTIBUS ARBITER FOR AM9516
PATTERN   AMP3-190
REVISION  01
AUTHOR    JOE BRCICH
COMPANY   ADVANCED MICRO DEVICES
DATE      11/15/87

CHIP MULTI2 PAL16R4

/BCLK /XACK  BRQ /BSY  /BPRN /DS    NC   /IORC /CS   GND
/OE   /RBEN /TBEN BACK /CEN  /BREQ /BUSY /BPRO /WAIT VCC

EQUATIONS

TBEN = IORC*CS

TBEN.TRST = /BACK

RBEN = /IORC*CS

RBEN.TRST = /BACK

WAIT =/XACK*BACK

BREQ := BRQ

BPRO = /BRQ*BPRN

/BACK := /BUSY

BUSY := BREQ*BPRN*/BSY*/BUSY
      + BREQ*BUSY*BPRN
      + DS*BUSY

CEN := BACK

;DESCRIPTION
;/CEN delays the commands to meet the multibus requirements that
;address and data be valid at least 50 ns prior to control active.
;If we do not allow preemption; remove BPRN from the second
;expression in the BUSY equation and eliminate the third expression
```

Figure 3. Source Listing for MULTIBUS to Am9516 Interface

Z-Bus and 8088/8086 Interface

This application note describes how replacing two 8086 support chips with a Z8000 support chip and a PAL16R8A allows the 8086 CPU to interface directly to the Z-BUS. Since the timing of the signals used is the same for the 8088 CPU, this circuit will work equally well in those applications.

Interfacing the 8086 CPU to the Z-BUS allows 8086 users to take advantage of the very powerful Z8000 peripheral and memory support circuits that are available. The Z8000 peripheral circuits in particular offer the user higher throughput rates, simpler control software and less system overhead requirements than any previous generation peripheral family for any CPU.

Design Requirements

The 8086 CPU can operate in two different modes. In minimum mode, it generates all the bus control and timing signals for the 8086 (8085, 8088) buses directly on-chip. In maximum mode, the CPU puts out status information early in each bus cycle and relies on an external bus controller chip, the 8288, to generate timing and control signals. This implementation uses the CPU in maximum mode and replaces the 8288 with a PAL16R8A that generates the Z-BUS timing and control signals from the status signals provided by the CPU. It also makes use of the AmZ8127 clock generator to allow precise timing resolution by providing an oscillator signal at 3 times the CPU clock frequency. The AmZ8127 provides all the clock generation functions of an 8284A as well as several additional functions. Either clock chip will work in this system.

The bus controller provides the following functions:

—Generates $\overline{AS}$, $\overline{DS}$, $\overline{INTACK}$ and R/W with proper timing relative to address and data.

—Provides simultaneous assertion of $\overline{AS}$ and $\overline{DS}$ during reset.

—Automatic insertion of 1 wait state for all I/O cycles.

—Synthesizes a single Z-BUS interrupt acknowledge cycle from the 8086 IACK cycles.

Figure 1 shows the circuit interconnection diagram. The system uses a high-speed PAL16R8A to generate $\overline{AS}$, $\overline{DS}$, R/W and $\overline{INTACK}$ of the Z-BUS, RDY for wait state generation, and three internal state variables. The registers are clocked with the 15-MHz OSC signal from the 8127. The five input signals to the PAL device are 5-MHz CPU Clock Signal (CLK), System Reset (RESET), and the three CPU Status States ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$).

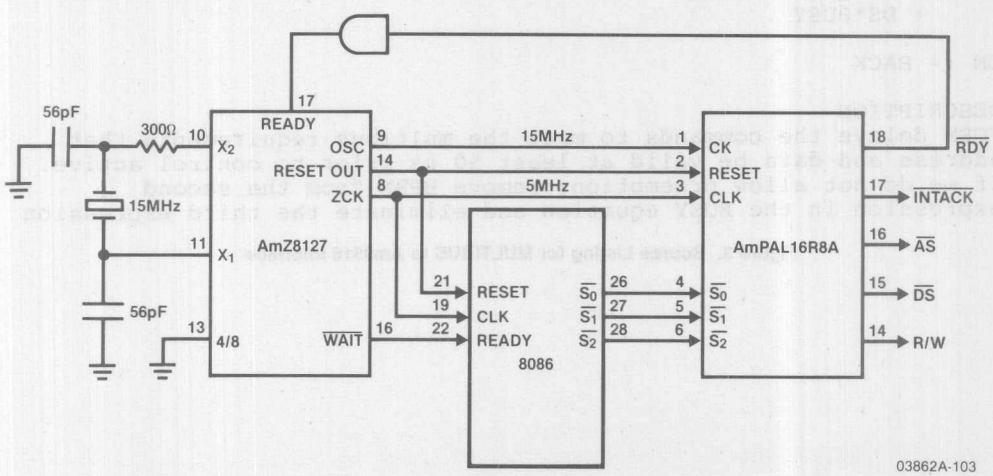


Figure 1. Circuit Interconnection Diagram

The CPU indicates the start of a bus cycle by bringing at least one of the status lines low from the idle high state (see Figure 2). This starts an internal timing sequence within the PAL device which corresponds closely to the various T states of a bus cycle. $\overline{AS}$ is asserted during the time CLK is LOW during T_1. $\overline{DS}$ is asserted at the start of T_3. If it is an I/O cycle, then RDY would be disabled for one CLK period straddling T_2 and T_3 causing the 8127 to request 1 wait state after T_3. In either case, $\overline{DS}$ remains asserted until after the first 1/3 of T_4, which is identified by the status lines returning to the idle state during the previous cycle. R/W is generated by sampling $\overline{S}_0$ and $\overline{S}_1$ during $\overline{AS}$.

It is in the realm of interrupts where the Z8000 peripherals shine over other peripherals. Each peripheral can identify many different exception conditions during its operation. The occurrence of one or more of these conditions causes activation of a single interrupt request line. The peripheral wants the CPU to respond with a single interrupt acknowledge cycle, during which the peripheral resolves priority and provides the CPU with enough status and vector information to allow it to respond to the exception without any further interrogation of the peripheral. This allows interrupt driven systems to achieve very high data throughput rates.

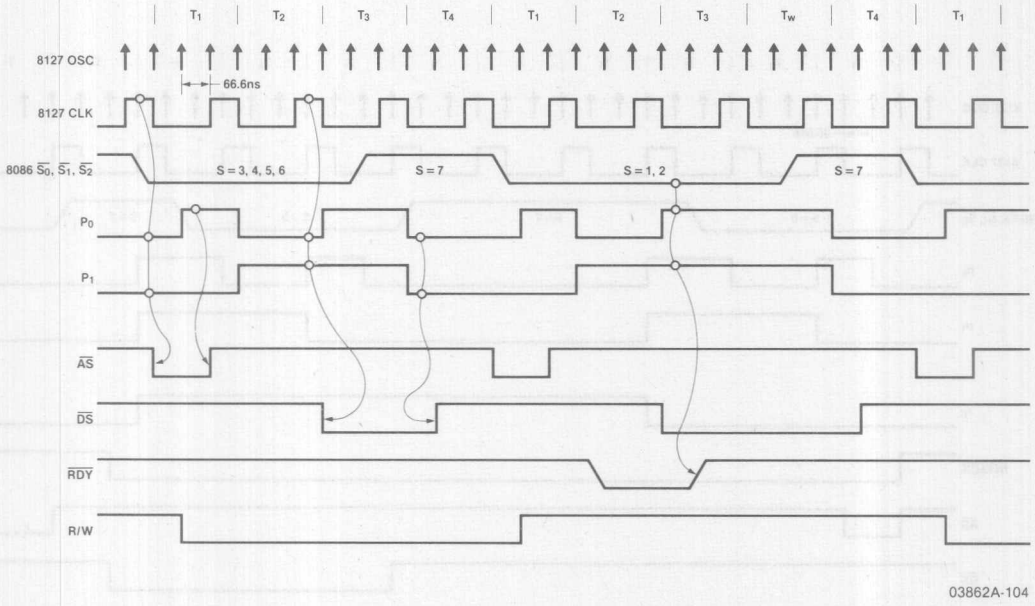


Figure 2. Memory and I/O Timing Diagram

The 8086 CPU responds to an interrupt request with a sequence of two interrupt acknowledge cycles, and only in the second is any data read off the bus. As stated before, the Z-BUS peripherals require only one acknowledge cycle. The timing of this has to be such that there is enough delay between $\overline{AS}$ going HIGH and $\overline{DS}$ going LOW to allow any prioritizing daisy chains to settle, and $\overline{DS}$ has to be wide enough to allow the peripheral time to place vector or status information on the bus. Figure 3 shows how these two requirements are accomplished by turning the two acknowledge cycles into one. The first cycle allows only $\overline{AS}$ and the second asserts only $\overline{DS}$ and does so for the complete cycle. This appears to the peripheral as one very long bus cycle which is identified as an interrupt acknowledge cycle by the assertion of $\overline{INTACK}$.

Design Approach

To implement this design in a PAL16R8A requires recognizing the Z-BUS timing characteristics in Figures 2 and 3. The major characteristic to consider is counting the phases of a bus cycle. Internal state variables P_0 and P_1 are the result (see Figure 4). An additional internal state variable (I_2) is necessary to count the second bus cycle of an interrupt acknowledge sequence. As shown in Figure 5, I_2 in conjunction with $\overline{INTACK}$ allows $\overline{AS}$ to be asserted only in the first interrupt acknowledge cycle and $\overline{DS}$ only in the second. The RESET input is used to initialize the internal variables and assert $\overline{AS}$ and $\overline{DS}$. Note also that $\overline{S}_0$ and $\overline{S}_1$ are included in the $\overline{DS}$ equation to prevent $\overline{DS}$ from being asserted during a halt cycle.

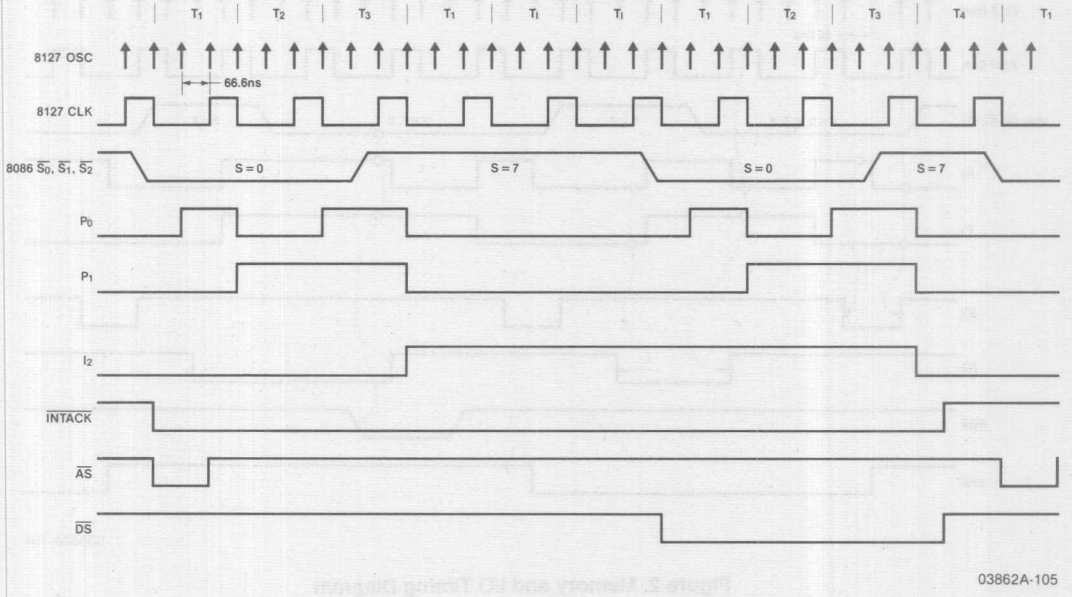


Figure 3. Interrupt Acknowledge Timing Diagram

The fact that PAL devices are user programmable allows a great deal of flexibility for the designer. Minor timing changes are easily implemented by simply adding or changing a term in the logic equations and reprogramming the device. In this system, we have timing resolution to 67 ns. This same configuration can be used with a 24-MHz crystal for 8-MHz CPU chips. The logic equations would change because the OSC period would be 42 ns. The only hardware change would be the crystal.

An additional PAL device could also perform chip select decoding based on both address and status signals.

Conclusion

We have seen how a properly programmed PAL device can be used to replace a specialized bus controller chip and allow an 8086 CPU to interface directly to Z-BUS peripheral(s) and/or memory systems. This brings all the advantages of the superior Z8000 peripheral family in terms of both throughput and ease of use to 8086 users with no increase in chip count while still allowing a wide range of design flexibilty. The PALASM software equations for the PAL16R8A 8086 to Z-BUS interface chip are shown in Figure 6.

P_1	P_0	PHASE	CPU T STATES
0	0	IDLE	T_4, T_1
0	1	$\overline{AS}$ TIME	T_1
1	0	$\overline{AS}$ TO $\overline{DS}$ DELAY	T_2
1	1	$\overline{DS}$ TIME	T_3, T_W

03862A-106

Figure 4.

I_2	$\overline{INTACK}$	$\overline{AS}$	$\overline{DS}$
NO	NO	YES	YES
NO	YES	YES	NO
YES	YES	NO	YES

03862A-107

Figure 5.

```
TITLE     8086 TO Z-BUS INTERFACE
PATTERN   AMP3-195
REVISION  01
AUTHOR    NICK ZWICK
COMPANY   ADVANCED MICRO DEVICES
DATE      11/15/87

CHIP ZBUS1 PAL16R8

CK   RESET CLK /S0 /S1 /S2  NC      NC    NC GND
/E /P0    /P1  /RW /DS /AS /INTACK RDY /I2 VCC

EQUATIONS

; INTERNAL STATE VARIABLES

P0 := /RESET* S0*/P0*/P1*/CLK
    + /RESET* S1*/P0*/P1*/CLK
    + /RESET* S2*/P0*/P1*/CLK
    + /RESET* P0*/CLK
    + /RESET* P1* CLK* S0
    + /RESET* P1* CLK* S1
    + /RESET* P1* CLK* S2

P1 := /RESET* P0*/P1* CLK
    + /RESET* P1*/CLK
    + /RESET* P1* CLK* S0
    + /RESET* P1* CLK* S1
    + /RESET* P1* CLK* S2

I2 := /RESET* INTACK*/I2* CLK* P0*.P1
    + /RESET* I2*/P1
    + /RESET* I2*/P0
    + /RESET* I2*/CLK* P0* P1

; Z-BUS OUTPUT SIGNALS

AS :=  RESET
    + /P0*/P1* CLK*/I2
    +  AS*/P0*/I2*/DS

DS :=  RESET
    + /INTACK*/P0* P1* CLK*S0
    + /INTACK*/P0* P1* CLK*S1
    +  I2* S0* S1* S2
    +  DS* P0* P1

RW :=  AS* S0*/S1
    +  RW*/AS

INTACK:= /RESET*AS*S0*S1*S2
       + /RESET*INTACK*/I2*P1
       + /RESET*I2

 /RDY := /RESET* S0*/S1* S2*/P0* P1
       + /RESET*/S0* S1* S2*/P0* P1
```

Figure 6. Source Listing for the Z-Bus to 8088/8086 Interface

VME Bus Control Simplified with PLDs

A High-Performance Bus

The VMEbus is a high-performance asynchronous bus capable of supporting multiple 16- and 32-bit processors. Its asynchronous nature permits easy implementation of hardware controllers that reduce the time required for bus control tasks such as bus arbitration. By offloading these tasks to hardware, bus arbitration time is minimized which consequently maximizes the overall bus transfer rate.

This article describes PLD implementations of two such controllers in a typical VMEbus system: the bus arbiter and the interrupter that comply with the VMEbus protocol. These state machines used to require multi-chip designs consisting of sequencers, microprogrammed control stores, and other MSI/LSI chips. By using the PAL22V10 programmable array logic device, you can reduce substantially the number of chips needed for these state-machine implementations.

Functional Modules

A typical VMEbus-based computer system contains a bus arbiter and an interrupt handler board. A processor/master initiates a data transfer to a slave by requesting control of the data bus from the bus arbiter. Once bus control has been granted to a master, control signals to and from the master and slave are exchanged which follow predefined protocols. These protocols guarantee an orderly transfer of data between the communicating modules. Interrupt servicing capability is provided by the interrupt handler board (see Figure 1).

Steps in Designing the Bus Controllers

VMEbus protocols outline the steps to perform any bus-related operation, such as transferring data over the bus between two modules. These protocols are described using flow diagrams with interactions between the communicating modules shown through various interface signals. For example, the priority bus arbiter flow diagram (Figure 2) shows how bus requests between two modules using the same request line are resolved.

The flow diagrams are analyzed to pick out the functions that can be incorporated into PLDs, and then state machines are created for these functions. These state machines can be described logically, or with flowchart symbols such as rectangular blocks (to symbolize events or control signals) and decision diamonds (to determine control program flow). Logic equations are then written for these state machines.

This methodical process is used in designing the two PLD-based VMEbus controllers: the bus arbiter and interrupt handler controller.

A PLD Bus Arbiter

Before any module can perform a data transfer, it must request control of the data transfer bus from the bus arbiter. This design uses a priority option bus arbiter implemented on a single PAL22V10 that monitors the four bus request lines BR0–3(L) with BR3(L) assigned the highest priority.

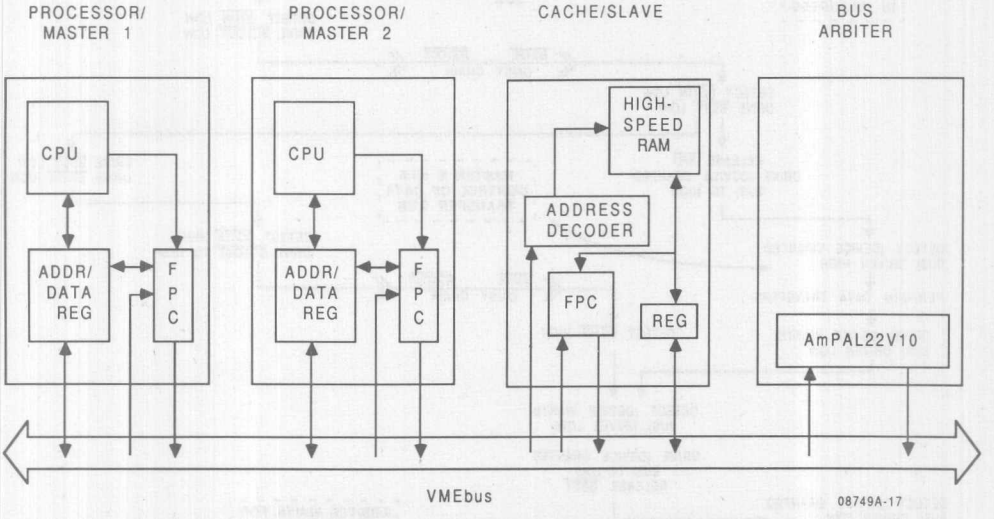


Figure 1. Intermodule Communication Through Am29PL141 (FPCs) and PLDs in a VMEbus System

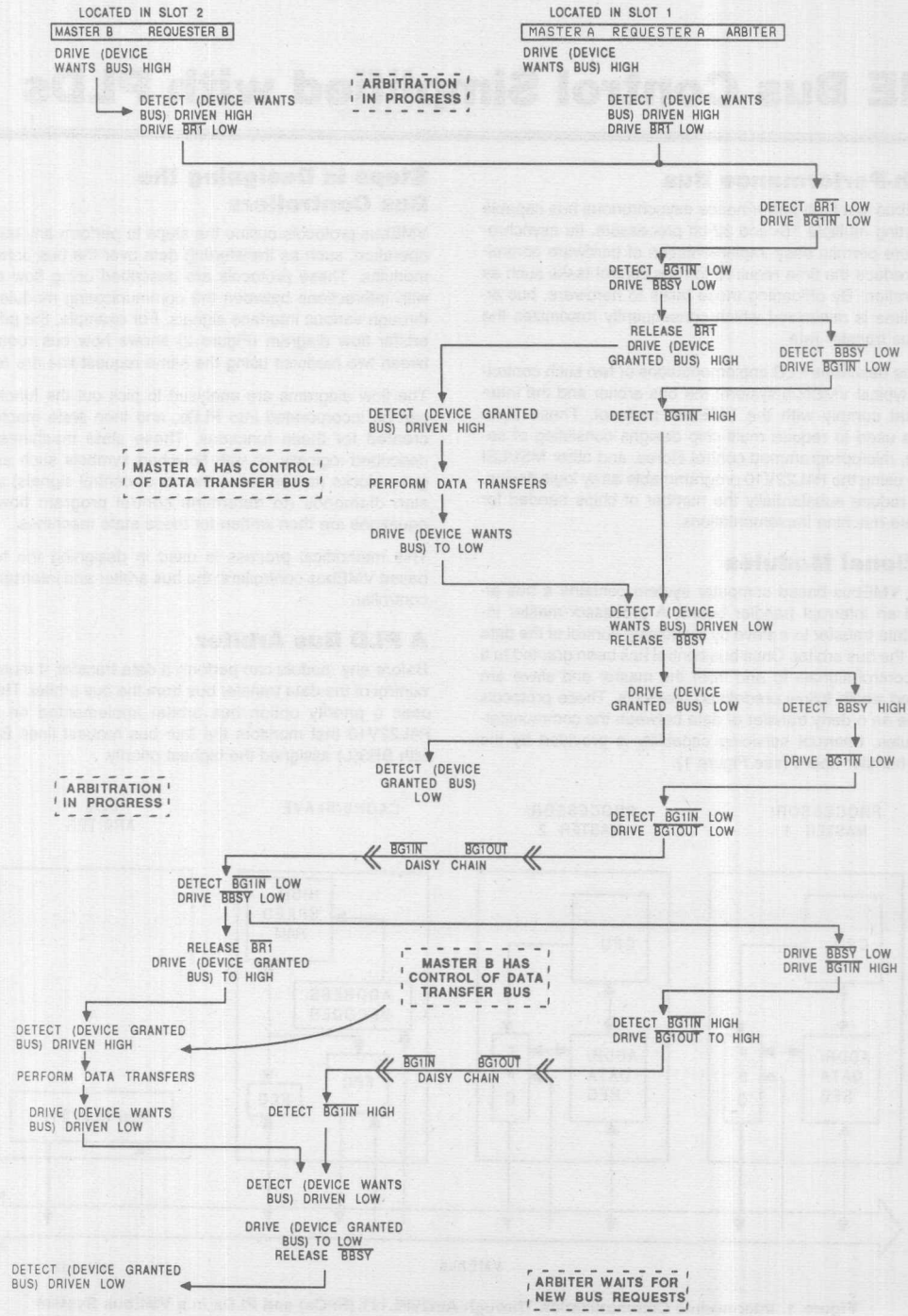


Figure 2. Bus Arbitration Flow Diagram

Based on the flow diagram (Figure 2), the arbiter grants the bus to the requesting module using the highest active request line, in this case BR1(L). A bus grant signal is daisy-chained to all the devices using this request line to resolve simultaneous bus requests from two or more modules. This dictates that the arbiter be in the first slot of the VMEbus system, and that the module physically closest to the arbiter through the daisy-chain will get the bus grant.

All bus requests are processed in parallel by the AND-OR array of the PLD. This allows priority arbitration to occur in a single clock cycle, which minimizes arbitration time and maximizes the data transfer rate on the VMEbus.

When the bus is free, the module using the highest active bus request line is granted the bus. This is described with Boolean logic notation where registered outputs are defined with ':=':

```
IF (/BBSY*(BR0+BR1+BR2      if bus not busy and a
+BR3)) THEN BEGIN             request line is active
  IF (BR3) THEN BG3IN := 1; if BR3 is active, grant
                              bus to device on BR3
  IF (/BR3*BR2) THEN        activate bus grant daisy
    BG2IN := 1;               chain 2
  IF (/BR3*/BR2*BR1) THEN   if BR1 is active and BR3
    BG1IN := 1;               and BR2 are not, then
                              grant bus to device
                              using request line 1
  IF (/BR3*/BR2*/BR1*BR0)   BR0–3 and BG0IN to
    THEN BG0IN := 1;          BG3IN are active low
  END ;
```

The PAL22V10 can be programmed with normal or inverted outputs and the logic software can support active high or low input polarities.

BG1IN(L) is asserted until the requesting module asserts BBSY(L). This is defined in logic equation form as:

```
IF (/BBSY*BG1IN) THEN       continue asserting
  BG1IN := 1;                 BG1IN until BBSY
                              becomes active
```

Priority Arbitration Options

It may be necessary for the arbiter to force the current bus master to relinquish bus control for certain conditions, such as if a higher priority operation must be attended to. This is done by asserting the bus clear signal BCLR(L) based on the priority of the current bus master and the bus clear conditions.

The PLD arbiter keeps track of the current bus master's priority by recording which bus request line was used to gain control of the bus. For example, two output registers called BUS_MASTER will be set to the binary value "2" if the current bus master used BR2(L) to gain control of the bus.

In this design, BCLR(L) wil be activated under two conditions:

1. If the BUS_MASTER is 2,1 or 0 and the active bus request line is 3 or 2
2. If BUS_MASTER is 0, and any bus request line is active.

When BUS_MASTER is 3, then the arbiter will not honor any bus requests until the bus busy line BBSY(L) is low. The above conditions are expressed logically as:

```
IF (BBSY*(BR0+BR1+BR2+BR3)) THEN
  BEGIN IF (BR3*(/MASTER[1:0] = 3) ) ) THEN
    BCLR := 1; "assert BCLR if MASTER < > 3"
  IF (/BR3*BR2*(/MASTER[1:0] = 3) ) ) THEN
    BCLR := 1;
  IF (MASTER[1:0] = 0) THEN
    BCLR := 1;
  END;
```

Once the BCLR(L) line is active, then it should be held until the current bus master releases BBSY(L). This is logically expressed as BCLR = BCLR*BBSY, or in a high-level syntax,

```
IF (BCLR*BBSY) THEN
  BCLR := 1;
```

The BCLR(L) signal is defined such that uninterruptible devices use BR3(L). Devices that can be temporarily suspended to accommodate interrupts and higher-priority operations are assigned to bus request line 0. The BCLR conditions will vary with your application, but any modifications will only require redefinitions of the high-level logic expressions.

The PAL22V10 is ideal as the bus arbiter because of the input/output signal requirements and the large number of product terms needed to logically define the bus grant and bus clear signals.

Processing Interrupts

When real-time or urgent response is needed by a processor, it generates an interrupt request signal and waits for the interrupt acknowledge signal IACK(L) and IACKIN(L) daisy-chain signal. A 3-bit value is then read from the VMEbus when the data strobes are active; this value indicates which interrupt request line was acknowledged. These three-bits are decoded to determine if it matches the processor's request level.

If the interrupt is acknowledged, then the interrupting processor puts its status or ID byte on the data bus for the interrupt handler. This data is used by the interrupt handler as an interrupt vector. This processor can then wait in a loop until the IACK(L) signal from the interrupt handler is driven HIGH to signify interrupt service completion.

Interrupt Handling: The Interrupt Handler Preprocessor (IHP)

To handle external I/O or special system events (e.g., timeout, overflow), interrupts are supported on the VMEbus through an interrupt handler (IH) module.

The logic complexity of the IH is reduced by offloading some of the initial interrupt recognition tasks to a PLD. A PAL22V10 can be programmed as an IHP to preprocess interrupt requests, obtain the bus and handle data transfers (such as interrupt acknowledge signals). Control is passed to the interrupt handler only when the IHP latches the interrupt vector.

Interrupt request processing, bus acquisition, and the interrupt vector transfer phase are all specified using logic equations which are written using a high-level Boolean notation. The PAL22V10 monitors seven interrupt request lines and five other control inputs, and generates ten registered outputs to send control signals to the interrupt handler and the VMEbus drivers.

Interrupt Logic

All interrupt request lines are monitored according to the following logic equation:

```
IF (IR1 + IR2 + IR3 + IR4 + IR5 + IR6 + IR7) THEN
  BR3 := 1 ;                    this interrupt handler uses
                                the BR3 request line
```

If any of the interrupt lines are active, then BR3(L) is asserted. This initiates the bus acquisition phase.

The next step is to wait for the bus grant in signal. Only when BGIN3(L) is active will BBSY(L) be active. This is expressed logically as:

```
IF (BR3*/BG3IN) THEN      —
   BR3 := 1;              —  ———————————  [A]
IF (BR3*BG3IN + BBSY*/SERVICE_DONE)
                                THEN ———  [B]
   BBSY := 1;
```

Equation [A] continually asserts BR3(L) as long as BG3IN(L) is not active, while [B] asserts BBSY(L) only when request line BR3(L) and BG3IN(L) are active, or if service has not been completed once the IHP asserts BBSY(L).

When the IHP is the bus master, it puts the 3-bit interrupt-line acknowledge value on the bus and applies the data strobes. When the DTACK(L) signal is received from the interrupter, the IHP then strobes the status byte from the bus into a register on the interrupt handler card. The IHP informs the interrupt handler that a status/ID byte is ready and to commence interrupt servicing.

The IHP takes only a single clock cycle to resolve interrupt priorities because all interrupt/input signal lines are processed in parallel by the PLD. This is expressed in a logic description language as follows:

```
IF (BBSY) THEN
   BEGIN
(1) IF (IR7) THEN                IR7 was active
      INTR[2:0] := 7;            3-bit value acknowledging
                                 interrupt line 7
    IF (/IR7*IR6) THEN           IR6 active, IR7 inactive
      INTR[2:0] := 6;            acknowledge interrupt line 6
    IF (/IR7*(IR6*IR5) THEN      IR5 highest priority
      INTR[2:0] := 5;            interrupt line active
                  :
                  :
                  :
(2) IF (/IR7*/IR6*/IR5*/IR4*/IR3*/IR2*IR1) THEN
      INTR[2:0] := 1 ;           acknowledge interrupt line 1
    END;
```

As noted above, if both the IR7(L) and BBSY(L) signal are active in (1), then regardless of the value of the other interrupt lines, the three-bit value generated will be "111" or 7. In (2), if IR1(L) and BBSY(L) are active and the other six control signals are inactive, then the three-bit output value will be "001" or 1. In (2), IR1(L) is the highest priority line active.

The remaining preprocessing involves completing the interrupt vector byte transfer. This is logically expressed as:

```
   IF (BBSY) THEN
     BEGIN
     IACK := 1;                  begin interrupt acknowledge
(X)  IF (DTACK) THEN             daisy chain; if device sends
       LATCH_STATUS := 1;        data transfer acknowledge
     END;                        (DTACK) signal, then latch
   IF (IACK) THEN                the status
     AS := 1;        assert the address strobe signal to inform
                     the interrupter that the interrupt
                     acknowledge level is ready
```

Once the 8-bit status of ID byte is transferred successfully via the DTACK(L) signal (X), the IHP PAL device informs the interrupt handler module to begin the interrupt service routine:

```
IF (BBSY*LATCH_STATUS +BBSY*START_SERVICE*
    /SERVICE_DONE) THEN
  START_SERVICE := 1;
```

In this design, the START_SERVICE signal is constantly asserted until the interrupt handler generates a SERVICE_DONE signal, at which point START_SERVICE is brought LOW. The above equations will generate the timing diagram in Figure 3.

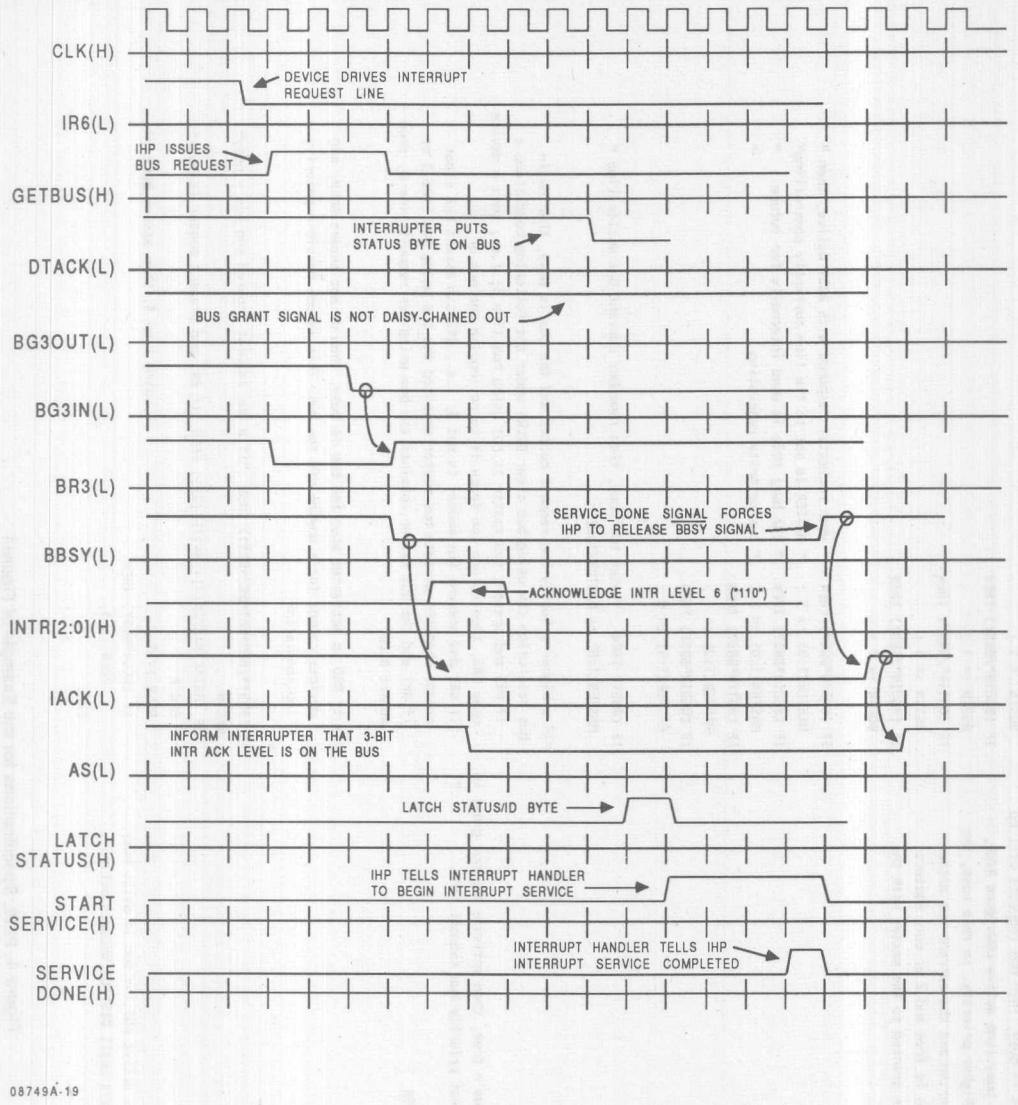


Figure 3. Timing Diagram

Controller Design Simplified with Development Tools

The task of programming FPCs and PLDs as VMEbus controllers is simplified with the development tools. The designer need only analyze the bus protocols, convert these into state machines and then write assembly language programs or high-level logic equations to describe these state machines. The assembler and logic software will then process these programs and equations to fit into the FPC or PLD.

```
DEVICE ARBITER (AMPAL22V10)  " VMEbus Priority Arbiter Option
For this application, if the bus is busy, then bus requests will be
deferred until the BBSY(L) (bus busy line active LOW) goes HIGH,
unless the bus request is of a higher priority. In this case, the
bus clear signal BCLR(L) is asserted and the current bus master
relinquishes control. If the bus is free and 2 or more masters
request service, then the bus is granted to the master with the
higher priority request line. "

PIN CLK         = 1
    /BBSY       = 2
    /BR0        = 3
    /BR1        = 4
    /BR2        = 5
    /BR3        = 6

    /BCLR       = 23
    /BG0IN      = 22
    /BG1IN      = 21
    /BG2IN      = 20
    /BG3IN      = 19
    MASTER[1:0] = 18,17;

BEGIN
"if a bus request occured and bus = free, then activate the bus grant in
 line corresponding to the highest priority bus request line          "

IF (/BBSY*(BR0+BR1+BR2+BR3)) THEN
  BEGIN
  IF (BR3) THEN
    BG3IN := 1 ;
  IF (/BR3*BR2) THEN
    BG2IN := 1 ;
  IF (/BR3*/BR2*BR1) THEN
    BG1IN := 1 ;
  IF (/BR3*/BR2*/BR1*BR0) THEN
    BG0IN := 1 ;
  END ;

" while the bus grant in line is active and the bus is still busy,
  then latch the bus grant signals until BBSY(L) becomes active"
IF (BG3IN*/BBSY) THEN
  BG3IN := 1 ;
IF (BG2IN*/BBSY) THEN
  BG2IN := 1 ;
IF (BG1IN*/BBSY) THEN
  BG1IN := 1 ;
IF (BG0IN*/BBSY) THEN
  BG0IN := 1 ;

IF (BG3IN*BBSY) THEN   " when requester responds with BBSY active, then "
  MASTER[1:0] := 3 ;   " MASTER is set to the line currently controlling"
IF (BG2IN*BBSY) THEN   " the bus; this is used internally for future    "
  MASTER[1:0] := 2 ;   " bus priority resolution                         "
IF (BG1IN*BBSY) THEN
  MASTER[1:0] := 1 ;
IF (BG0IN*BBSY) THEN
  MASTER[1:0] := 0 ;

IF (BBSY) THEN   " when bus busy, then remember current bus master line "
  MASTER[1:0] := MASTER[1:0] ;

"if a higher priority bus request occurs and the bus is busy, then begin
 bus resolution (by using bus clear BCLR) under the following conditions :
 - if BR3 and present bus master is not using bus line 3; i.e., once a device
   uses BR3, then no one can force it to relinquish the bus or
 - if BR2 and present bus master is not 3; i.e., BR2 can assert bus clear
   except when the presetn bus master obtained the bus using bus line 3 or
 - if BR1 and the bus master obtained the bus using bus request line 0, then
   assert BCLR

==> if BR0 is activated when the bus is busy, then do not assert BCLR; BR0
    devices cannot force anyone off the bus. This is one possible priority
    implementation.
"
IF (BBSY*(BR0+BR1+BR2+BR3)) THEN  "if a bus request occured and bus = busy"
  BEGIN
  IF (BR3*(/MASTER[1]+/MASTER[0])) THEN "if BR3 and present master is not "
    BEGIN
    BCLR := 1 ;                          " using line 3, then assert BCLR  "
    IF (BCLR*BBSY) THEN
        BCLR := 1;
    END;
```

Figure 4. PLPL Specifications for the Example of Figure 1

```
    BCLK := 1 ;
    IF (BCLR*BBSY) THEN
        BCLR := 1;
    END;
  IF (/BR3*/BR2*BR1*/MASTER[1]*/MASTER[0]) THEN
    BEGIN
    BCLR := 1 ;
    IF (BCLR*BBSY) THEN
        BCLR := 1;
    END;
  END ;
END.
"Test vectors used for simulation and testing"
TEST_VECTORS
IN  CLK , /BBSY , /BR0 , /BR1 , /BR2 , /BR3 ;
OUT /BCLR , /BG0IN , /BG1IN , /BG2IN , /BG3IN , MASTER[1:0] ;
BEGIN
"   / / / / /     / / / / /   "
" c b b b b b     b b b b b m m"
" l s r r r r     c g g g g s s"
" k y 0 1 2 3     r 0 1 2 3 1 0"
"-------------------------------"
  C 1 0 1 1 1     X X X X X X X ;
  C 0 1 1 1 1     X X X X X X X ;
  C 1 1 1 1 1     X X X X X X X ;
  C 1 1 0 0 1     X X X X X X X ;
  C 0 1 1 1 1     X X X X X X X ;
  C 0 1 1 1 0     X X X X X X X ;
  C 0 1 1 1 0     X X X X X X X ;
  C 0 1 1 1 0     X X X X X X X ;
  C 1 1 1 1 0     X X X X X X X ;
  C 1 1 1 1 0     X X X X X X X ;
  C 1 1 1 1 0     X X X X X X X ;
  C 0 1 1 1 1     X X X X X X X ;
  C 0 0 1 1 1     X X X X X X X ;
  C 0 0 1 1 1     X X X X X X X ;
  C 0 1 0 1 1     X X X X X X X ;
  C 0 1 1 0 1     X X X X X X X ;
  C 0 1 1 1 0     X X X X X X X ;
END.
```

Figure 4. PLPL Specifications for the Example of Figure 1 (Cont'd.)

```
DEVICE INTR_HND_CTRLR (AMPAL22V10) " VMEbus interrupt handler preprocessor

This PAL serves as an interface between the interrupt servicing logic in the
interrupt handler and the bus arbiter: when an interrupt is detected, the
PAL requests the bus from the arbiter. When the bus is granted, the bus
busy BBSY(L) is asserted. The 3 bit code corresponding to the highest
priority interrupt request is put on the bus, and both IACK(L) and AS(L)
are also asserted. The PAL waits for DTACK(L) LOW and then latches the
status/ID byte. The START_SERVICE signal informs the interrupt handler
to begin the interrupt service sequence with the status/ID byte latched in.
Note : this interrupt handler interface PAL handles all interrupts using
        bus request line BR3; this is application-dependent "
PIN
 CLK    = 1
 /IR1   = 2            "interrupt request lines 1 to highest priority 7"
 /IR2   = 3
 /IR3   = 4
 /IR4   = 5
 /IR5   = 6
 /IR6   = 7
 /IR7   = 8
 /DTACK = 9            "data transfer ack; used by interrupter to indicate
                        status/ID data byte ready on bus "
 /BG3IN = 10           "bus grant in signal from arbiter"
 SERVICE_DONE = 13     "signal from interrupt handler indicating
                        service complete"
 LATCH_STATUS  = 23    "latches data from bus for interrupt service routine"
 /BBSY  = 22           "bus busy"
 /IACK  = 21           "interrupt acknowledge"
 /AS    = 20           "address strobe"
 /BR3   = 19           "bus request signal to bus arbiter"
 GET_BUS       = 18    "used internally by PAL "
 INTR[2:0]     = 15:17 "3-bit level code put on bus   "
 START_SERVICE = 14 ;  "sent from PAL to intr handler "

BEGIN

"if any interrupt request line is active, then try getting the bus. All the
 interrupts handled by this PAL will use the bus request line BR3. A
 request can occur only when no other interrupt is being serviced (i.e.,
 bus should not be busy and bus grant in should not be active) "
IF (/BBSY*/BG3IN*(IR1 + IR2 + IR3 + IR4 + IR5 + IR6 + IR7)) THEN
  BEGIN
  GET_BUS := 1 ;
  BR3 := 1 ;        "subject to application"
  END ;

"if interrupt handler wants the bus, the bus grant line is still inactive,
 and the interrupt request is still active, then continue requesting the bus "
IF (GET_BUS*BR3*/BG3IN*(IR1 + IR2 + IR3 + IR4 + IR5 + IR6 + IR7)) THEN
  BEGIN
  BR3 := 1 ;
  GET_BUS := GET_BUS ;
  END;

"if bus grant in line is active, then assert the bus busy signal"
IF (GET_BUS*BG3IN + BBSY*/SERVICE_DONE) THEN
  BBSY := 1 ;

"if interrupt handler received bus, then acknowledge the interrupt"
IF (BBSY) THEN
  BEGIN
  IF (/START_SERVICE) THEN
    IACK := 1 ;
  IF (DTACK) THEN          "when the interrupter responds with DTACK(L), then  "
    LATCH_STATUS := 1 ;    "latch the status/ID byte into the interrupt handler"

  IF (IR7) THEN
    INTR[2:0] := 7 ;      "IR7 line is highest priority"
  IF (/IR7*IR6) THEN
    INTR[2:0] := 6 ;
  IF (/IR7*/IR6*IR5) THEN
    INTR[2:0] := 5 ;
  IF (/IR7*/IR6*/IR5*IR4) THEN
    INTR[2:0] := 4 ;
  IF (/IR7*/IR6*/IR5*/IR4*IR3) THEN
    INTR[2:0] := 3 ;
  IF (/IR7*/IR6*/IR5*/IR4*/IR3*IR2) THEN
    INTR[2:0] := 2 ;
  IF (/IR7*/IR6*/IR5*/IR4*/IR3*/IR2*IR1) THEN
    INTR[2:0] := 1 ;      "IR1 line is lowest priority"
  END;
```

Figure 5. PLPL Specifications for the Example of Figure 1

```
IF (IACK*/AS + IACK*/DTACK) THEN  "hold the interrupt acknowledge level until"
  INTR[2:0] := INTR[2:0] ;        "strobed by address strobe signal or until
                                   the interrupter acknowledges the 3-bit
                                   level with DTACK                          "

   "if the IACK line is active and interrupt servicing has not yet begun
    activate the address strobe line AS "
IF (IACK*/START_SERVICE) THEN
   AS := 1 ;

"send the start service signal until service is done"
IF (BBSY*LATCH_STATUS + BBSY*START_SERVICE*/SERVICE_DONE) THEN
  START_SERVICE := 1 ;

END.

TEST_VECTORS
IN   CLK , /IR1 , /IR2 , /IR3 , /IR4 , /IR5 , /IR6 , /IR7 ,
     /DTACK , /BG3IN , SERVICE_DONE;
OUT  START_SERVICE , GET_BUS , INTR[2:0] , LATCH_STATUS ,
     /AS , /IACK , /BBSY , /BR3 ;
BEGIN
" C / / / / / / / / / S   |  S G I I I L / / / / "
" L I I I I I I I D B R   |  R E N N N T A I B B "
" K R R R R R R R T G V   |  V T T T T C S A B R "
"   1 2 3 4 5 6 7 A 3 D   |  C B R R R H   C S 3 "
"               C I N     |  E U 2 1 0     K Y   "
" _____________K_N_E___|____S_____________"
   C 1 1 1 1 0 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 0 1 1 1 0 0      X X X X X X X X X ;
   C 1 1 1 1 0 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 0 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 1 1 0      X X X X X X X X X ;
   C 1 1 1 1 1 1 1 1 1 1      X X X X X X X X X ;
END.
```

Figure 5. PLPL Specifications for the Example of Figure 1 (Cont'd.)

IMPORTANT
NOTES

Communications

Introduction to Communication Systems

Modern communications has developed from two major sources: telephones and computers. Within telephony, what was once straightforward engineering in the analog telephone system has grown over 100 years into the sophisticated discipline of telecom. Within computers, communication needs have expanded rapidly over the last 20 years from simple asynchronous terminals clustered around a host to also include networking of PCs, workstations and minicomputers across rooms and continents. Both areas can be expected to merge into complex information networks which handle all aspects of communications over the next 10–20 years. A graphic portrayal of a hypothetical future system is shown in Figure 1.

Historically, the system architecture, engineering and components of these two converging areas have been dissimilar. As telephony systems become digital-based and computer systems use more of the telephony system for interaction, this dissimilarity is eroding. In particular, digital design techniques developed in computer engineering are used increasingly in telecom. The original application was in the trunk links between exchanges where digital time division multiplexing allowed a much higher density of circuits over a given pair of wires. This spread into the switch, line card and other areas of the exchange until only the subscriber line to the exchange remains largely analog. Development of the Integrated Services Digital Network (ISDN) international standard will make this digital as well, enabling data and video as well as voice to be handled in a uniform and flexible manner. Future telecom engineering will therefore have much in common with the digital design techniques used in computers and the distinction between them will become minimal.

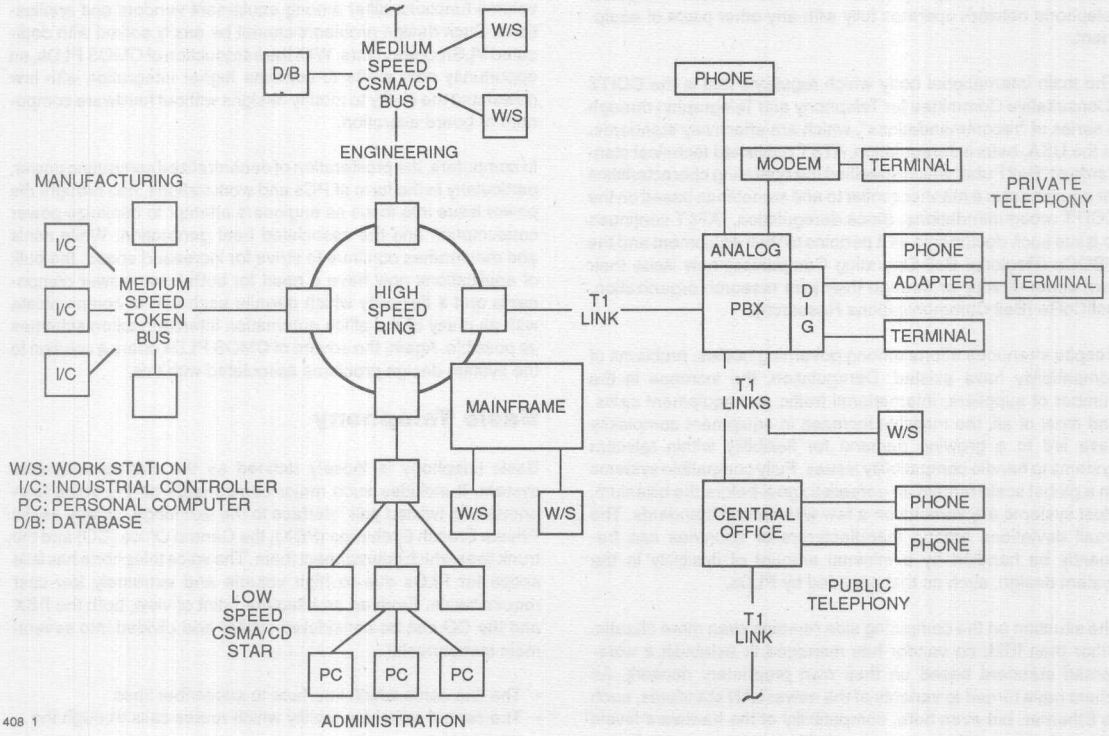


Figure 1. Communications in the Nineties

Along with the strong trend in equipment design towards digital systems, there is a parallel move towards a more global market for equipment. The development costs for major pieces of telecom systems, such as central office switches, require a large market in order to amortize the costs. This implies a requirement for standards beyond national boundaries, permitting sales into foreign markets with minimum engineering changes. This increased market size on top of increased complexity of services is realizable only through widespread standardization. It also makes such a universal system more complex and therefore more elusive.

The computer area has evolved around a number of proprietary computer families whose architectures and operating systems were incompatible to a large degree. The need to interconnect multiple computer systems with high-speed links led to a polarization around proprietary standards as local area network (LAN) interconnect standards were evolved by each major manufacturer. Recent moves towards decentralized computing, based on personal workstations of varying power, has crystallized users around a few LAN standards, but the dream of universal LAN interconnectivity among equipment is still far off.

Communications therefore requires widespread standards for any meaningful inter-vendor or international operation. Telephony is the more advanced in this area. Standards efforts by the Bell system in the US and the local PTT (the governmental Post, Telephone and Telegraphy authority) in other industrial countries has ensured that any equipment which connects to the public telephone network operates fully with any other piece of equipment.

The main international body which regulates that is the CCITT (Consultative Committee for Telephony and Telegraphy) through a series of "recommendations", which are effectively standards. In the USA, before deregulation, AT&T published technical standards or "Bell Pubs" which specified the operating characteristics for equipment in a manner similar to and sometimes based on the CCITT recommendations. Since deregulation, AT&T continues to issue such documents as it pertains to their equipment and the RBOCs (Regional Bell Operating Companies) now issue their own either directly or through their joint research organization, BellCoRe (Bell Communications Research).

Despite strenuous efforts among governing bodies, problems of compatibility have existed. Deregulation, the increase in the number of suppliers, international traffic and equipment sales, and most of all, the massive increase in equipment complexity have led to a growing demand for flexibility within telecom systems to handle compatibility issues. Fully compatible systems on a global scale remain an unrealistic goal before the billenium. Most systems are variants on a few widespread standards. The small deviations among manufacturers or countries can frequently be handled by a minimal amount of flexibility in the system design, such as that provided by PLDs.

The situation on the computing side remains even more chaotic. Other than IBM, no vendor has managed to establish a widespread standard based on their own proprietary network. All others have turned to variants of the newer LAN standards, such as Ethernet, but even here, compatibility at the hardware levels does not necessarily mean compatibility at the upper or software levels.

Features of Programmable Logic in Communications

Programmable logic has already demonstrated its usefulness in computer systems in a variety of roles, including address decoding, state machines and "garbage collection". All these applications exist in communications, but others particular to this market also exist.

The problems of interconnecting dissimilar systems will remain in communications for a number of years. Equipment manufacturers are therefore faced with a dilemma of how to combine the economies of scale associated with volume production with the need to interface with multiple vendor equipment, some of which deviates from the standards. PLDs offer a unique method of customizing logic to conform to deviations while maintaining identical hardware layout and parts.

Power and space requirements are considerations in most computer systems, but normally are relegated to being secondary issues when compared to performance. The opposite is true of telephony, where these factors assume paramount importance over speed. In order to satisfy requirements from both areas, components must therefore be available in both high-speed and low-power versions.

In telephony, certain high-volume functions such as line card interfaces justify VLSI devices. But a large number of other lower-volume functions differ among equipment vendors and applications. Such design problems cannot be easily solved with dedicated VLSI components. With the introduction of CMOS PLDs, an opportunity now exists to combine higher integration with low power and the ability to modify designs without hardware component or board alteration.

In computers, the proliferation of decentralized computing power, particularly in the form of PCs and workstations, has brought the power issue into focus as engineers attempt to minimize power consumption and the associated heat generation. While minis and mainframes continue to strive for increased speed, the bulk of applications now have a need for both lower-power components and a flexibility which permits each unit to communicate with as many of the office automation interconnection schemes as possible. Again, the advent of CMOS PLDs offers a solution to the system design problems associated with this.

Basic Telephony

Basic telephony is loosely defined as the classic telephone system. It includes such major components as the voice telephone with twisted pair interface to the exchange, as well as the Private Branch Exchange (PBX), the Central Office (CO) and the trunk lines which interconnect them. The voice telephone has little scope for PLDs due to high volume and extremely low cost requirements. From an architecture point of view, both the PBX and the CO can be considered similar and divided into several main components:

- The line cards which interface to subscriber lines
- The central switching facility which routes calls through the exchange
- The trunk interface which connects to other exchanges

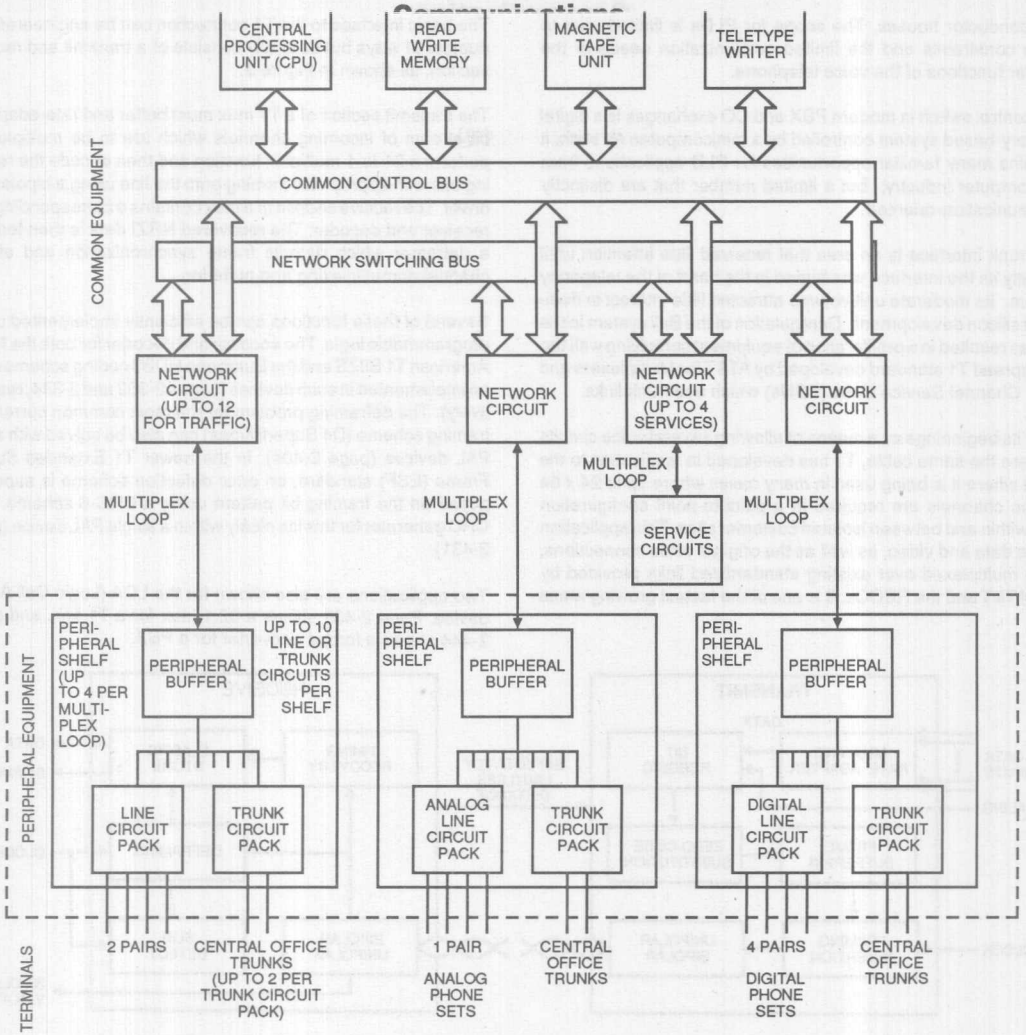


Figure 2. PBX Architecture Example

408 2

Line cards are the most cost- and power-sensitive components of the basic telephony system. They are usually fabricated in volume and incorporate VLSI devices available from several semiconductor houses. The scope for PLDs is limited, due to these constraints and the limited customization needs of the simpler functions of the voice telephone.

The central switch in modern PBX and CO exchanges is a digital memory-based system controlled by a minicomputer. As such, it contains many familiar opportunities for PLD applications from the computer industry, but a limited number that are distinctly communication-oriented.

The trunk interface is an area that received little attention until recently as the interface was buried in the heart of the telephony system. Its moderate unit volume attracted little interest in dedicated silicon development. Deregulation of the Bell system in the US has resulted in a proliferation of equipment complying with the widespread T1 standard developed by AT&T for Multiplexers and other Channel Service Units (CSUs) which use trunk links.

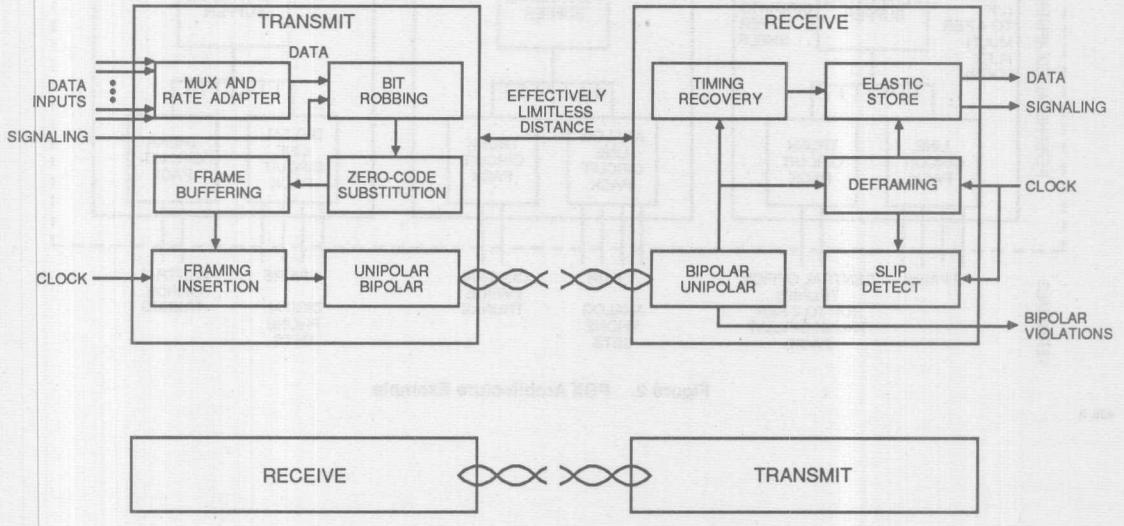
From its beginnings as a means of allowing several voice circuits to share the same cable, T1 has developed in application to the stage where it is being used in many cases where up to 24 x 64 Kb/sec channels are required in a point-to-point configuration both within and between isolated customer sites. This application allows data and video, as well as the original voice connections, to be multiplexed over existing standardized links provided by both AT&T and the RBOCs. It is one of the fastest growing areas of modern telecom and is spawning related equipment such as T1 switching nodes and T1 testers.

The basic interface to the T1 connection can be engineered in a number of ways but basically consists of a transmit and receive section, as shown in Figure 3.

The transmit section of a T1 mux must buffer and rate-adapt the bit stream of incoming channels which are to be multiplexed, perform a 24-to-1 multiplex function and then encode the resulting bits with appropriate framing onto the line using a bipolar line driver. The receive section of a mux contains a corresponding line receiver and decoder. The recovered NRZI data is then fed into a deframer which detects frame synchronization and allows channel demultiplexing and buffering.

Several of these functions can be efficiently implemented using programmable logic. The encoder and decoder for both the North American T1 B8ZS and the European HDB3 coding schemes can be implemented in such devices (pages 2-362 and 2-384, respectively). The deframing problem for the most common current T1 framing scheme (D4 Superframes) can also be solved with a few PAL devices (page 2-404). In the newer T1 Extended Super-Frame (ESF) standard, an error detection scheme is superimposed on the framing bit pattern using a CRC-6 scheme. The CRC generator for this fits nicely within a single PAL device (page 2-431).

Two applications are also shown for the LCA (Logic Cell Array) device. Page 2-435 shows a 32:1 mux for a T1 link, and page 2-444 shows a format converter for a PBX.



408 3

Figure 3. Basic T1 Connection

Data Communications

Unlike telephony, data communications is a relatively new discipline and equipment designed to interconnect data processing equipment evolved in parallel with mainframe computers. Such equipment tends to be high-speed and less concerned with power requirements than telephony equipment. Its main function is to transmit binary data in a reliable manner across a room or across a continent between computers compatible enough to exchange data. A wide variety of methods exist to do this at speeds ranging from 300 bits per second to 100,000,000 bits per second. As distance increases, the speed options available become limited. Generally, the options can be considered in a number of categories, in order of increasing speed, cost and sophistication:

- Asynchronous terminals and connections using existing analog phone lines
- Synchronous terminals and dedicated data links
- High-speed point-to-point links, including microwave and satellite
- Multipoint, medium-speed local area networks
- Multipoint backbone, high-speed local area networks

The slowest and most widely available category includes most "dumb" terminals, which are normally linked to a host system either directly via an RS-232C/V.24 serial link or over a standard telephone line via a voiceband modem. Maximum data rates available are 9600 bits/sec and the typical modem rates are currently 300 and 1200 baud. Devices used in building either type of link are typically LSI low-cost NMOS devices with most new devices being VLSI CMOS. Where PAL devices offer the greatest advantage is in higher-performance applications for which VLSI solutions do not yet exist.

There are, however, a number of additional control functions required around standard UART/USARTs which can be optimally implemented in a single PAL device, such as providing address latching and an asynchronous divider around a Motorola 6850 Asynchronous Communications Interface Adapter (page 2-453).

In the case of modems, however, several highly sophisticated techniques beyond that found in the standard modem "box" can be used to improve throughput on the link. One technique used to increase the bit rate across the link both in the slow, voiceband modems and in the higher-speed satellite modems is Quadrature Amplitude Modulation (QAM). Another technique uses a convolutional encoding scheme to increase the reliability of data transmitted across a noisy link. This cuts down the requirement for retransmission of lost data and effectively increases the throughput of the link.

Both schemes can be implemented efficiently in PAL devices as described in the application notes on pages 2-456 and 2-463.

Synchronous terminal links usually comply with one of a number of standards for which low-cost LSI devices exist. These provide for data rates above that of async links and generally can perform up to 1 Mbit/sec and beyond, but only over a limited distance, such as within a building. Data rates above 2400 baud over long distances are currently expensive and limited in popularity.

One scheme which is expected to overcome this and gain wide application over the coming years is the provision of 64Kbit/sec data across the new digital telephone system which will be available on the international Integrated Services Digital Network (ISDN) "S" and "U" interfaces to the phone line. This is a standard to which virtually every telephony authority and a large number of major computer manufacturers have given their support. Among the difficulties which will be encountered is the problem of adapting existing terminal devices to communicate with the wave of VLSI CMOS devices which are being released for use on the ISDN "S" and "U" interface link.

The "S" interface is being addressed by a number of VLSI solutions. The recently standardized "U" interface has no such devices and has an additional encoding/decoding requirement similar to trunk lines, due to the longer distances involved to the central office. A 4B3T coding scheme is used, which can be implemented using PLDs, as shown in the application note on page 2-475.

Further applications in data communications can be found in LANs. While several VLSI chip sets exist for standards such as Ethernet, a number of proprietary systems are in use and their numbers are increasing steadily. Some of the more basic functions required in the LAN interface, such as the Manchester encoding function of Ethernet, can be provided in a single PAL device (page 2-499).

Conclusion

The opportunities for the use of PLDs in communications is widespread and is becoming increasingly important for the following reasons:

- Power and space limitations.
- Increasing adherence to multiple standards, yet the need for...
  - Standardized hardware to permit low-cost volume production.
  - Need to reconfigure equipment in the field at the minimum overhead.

Where speed is an issue, the broad range of fast bipolar PAL devices provides a range of solutions. But in the more recent trend towards low-power circuitry, the need for which is most apparent in equipment such as in fanless or portable personal computers, The family of low-power CMOS devices provides the same functionality at a fraction of the power budget.

B8ZS Coding Using CMOS ZPAL™ Devices

AN-169

Bipolar with eight zero substitution (B8ZS) coding provides clear channel capability, which is required in an Integrated Service Digital Network (ISDN) or an open network. A (B8ZS) encoder and decoder can be implemented in CMOS PAL devices. The encoder and decoder convert between *non-return-to-zero* (NRZ) data and B8ZS while providing a flexible interface to framing chips.

Principles of Coding

Communication lines require a data stream with imbedded clocking information. T1 digital communication lines transmit Alternate Mark Inversion (AMI) coded voice and data. For AMI, logic "1's" are encoded as alternating positive and negative pulses, while logic "0's" are encoded as zero voltage. This coding scheme ensures that consecutive pulses alternate in polarity. The pulse edges provide the clocking. A pair of consecutive pulses with the same polarity is called a bipolar violation.

Since a string of zeros in AMI coding does not contain any transitions, a clock cannot be derived. If this string of 0's is too long, the system will lose synchronization. Therefore, the AT&T T1 standard limits the maximum number of consecutive 0's to fifteen. For voice transmission this restriction is acceptable; however, for data it is not. A variation of AMI could provide clocking information while not restricting user data.

B8ZS coding is such an improvement on the AMI code; logic "1's" are encoded as alternating positive and negative pulses, while logic "0's" are encoded as zero voltage. Additionally, B8ZS substitutes a string of eight 0's with a code word as shown in Table 1. The polarity of the preceding pulse determines which B8ZS code word is substituted. The B8ZS code words contain bipolar violations in the fourth and seventh bit positions.

Since B8ZS coding allows strings of 0's for any length, it supplies clear channel capability. Clear channel capability is required for an open network and ISDN. With B8ZS coding, the maximum length of consecutive zeros is seven bits.

PRECEDING PULSE	B8ZS CODE WORDS
+	000+-0-+
-	000-+0+-

Table 1. B8ZS Coding

B8ZS coding is used in the following applications:

- T1 multiplexers
- T1 repeaters
- T1 channel banks
- T1 network equipment
- Video CODECs

Functional Description

Implementing B8ZS coding in such equipment requires converting between NRZ and B8ZS format. In this design, CMOS PAL devices encode and decode data while providing flexibility. Current single chip solutions do not interface to all framing devices. Using the PAL devices, B8ZS line coding can be added to any framing technique. This design performs two operations: encoding and decoding. The encoding operation converts NRZ data into positive and negative pulses with B8ZS code words. Decoding converts the positive and negative pulses into NRZ data and substitutes eight 0's for the B8ZS code words shown in Table 1.

B8ZS Encoder

The B8ZS encoder converts NRZ data to B8ZS data. Each string of eight consecutive 0's is replaced with a B8ZS code word.

Two PAL devices can be configurated to encode the data stream. Figure 1 shows the PAL devices, designated, PAL_A and PAL_B. Signals on Figure 1 are explained in Table 2. PAL_A, a PALC16R8Z, implements the 3-bit counter, which searches for strings of eight 0's. An 8-bit string of 0's is detected by C2-C0 and NRZ_IN. PAL_A also incorporates a 5-stage shift register, which delays the NRZ_IN data stream to ensure proper alignment with the counter's output. The delayed data, NRZ_DELAY, is encoded by PAL_B.

PIN	DESCRIPTION
$\overline{RST}$	Active low master reset
NRZ_CK	External NRZ clock
NRZ_IN	NRZ input data stream
C(2-0)	Counter output
NRZ_DELAY	Delay NRZ data stream
CS	Current state
PPO	Positive pulse output
NPO	Negative pulse output

Table 2. B8ZS Encoder Pin Description

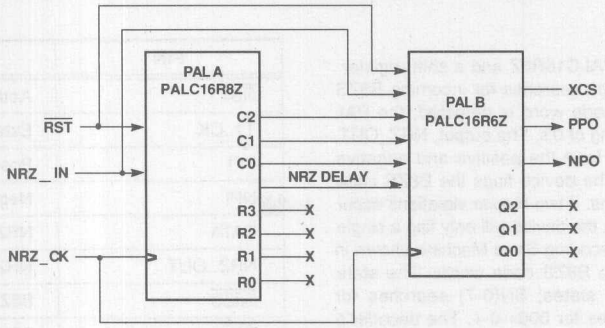


Figure 1. B8ZS Encoder

PAL_B, a PALC16R6Z, encodes the data using two state machines; one generates the B8ZS sequence, and the other generates the positive and negative pulse outputs. Figure 2a shows the B8ZS Sequence State Machine, which generates the B8ZS sequence. If an 8-bit string of 0's is detected, the B8ZS Sequence State Machine forces the B8ZS code word to be encoded. Otherwise, B8ZS Encoder State Machine performs normal AMI-type encoding. The B8ZS Encoder State Machine encodes NRZ_DELAY into positive and negative pulses based on the B8ZS Sequence states, Q2-Q0 (see Figure 2b). The positive and negative pulses (PPO and NPO) occur six clock cycles after the input NRZ_IN.

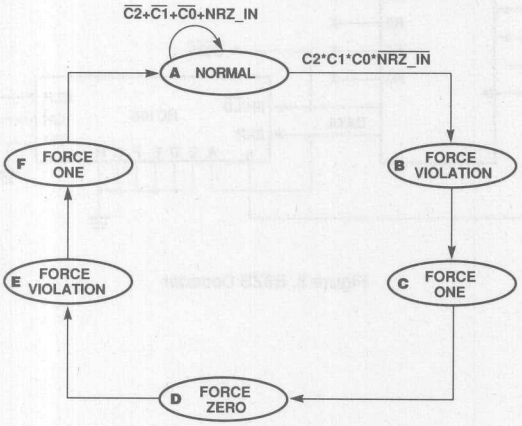


Figure 2a. B8ZS Sequence State Machine

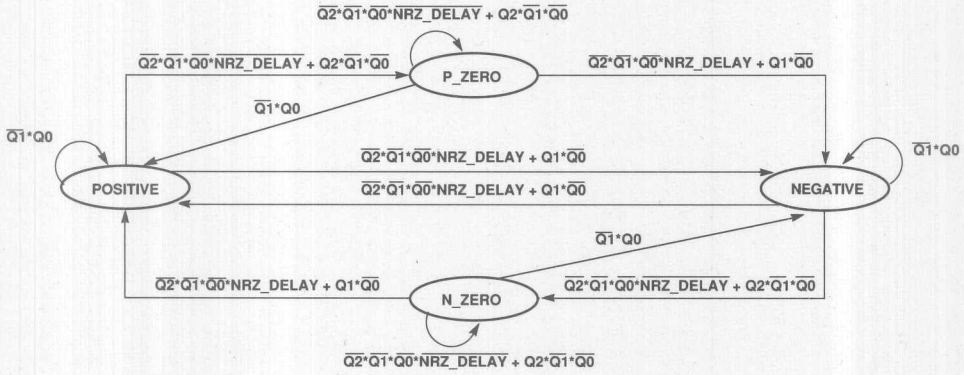


Figure 2b. B8ZS Encoding State Machine

B8ZS Decoder

Decoding is performed by a PALC16R6Z and a shift register (see Figure 3). The PAL device searches for incoming B8ZS code words. When a B8ZS code word is detected, the PAL device substitutes an 8-bit string of 0's. The output, NRZ_OUT, is delayed eight clock cycles from the positive and negative pulse inputs (PPI and NPI). The device flags the B8ZS code words and any bipolar violations. If two bipolar violations occur in a non-B8ZS 8-bit data string, the device will only flag a single bipolar violation. The B8ZS Decoding State Machine shown in Figure 4 searches for the two B8ZS code words. The state machine has two groups of states; SN(0-7) searches for 000-+0+- and SP(0-7) searches for 000+-0-+. The decoder's state machine shows reduced equations from the B8ZS and BPV (Bipolar Violation) outputs. Note: PPI and NPI are never both HIGH.

PIN	DESCRIPTION
$\overline{RST}$	Active low master reset
T1_CK	External T1 clock
PPI	Positive pulse input
NPI	Negative pulse input
DATA	NRZ data stream
NRZ_OUT	NRZ output data stream
$\overline{B8ZS}$	B8ZS flag
$\overline{BPV}$	BPV (Bipolar Violation) flag

Table 3. B8ZS Decoder Pin Description

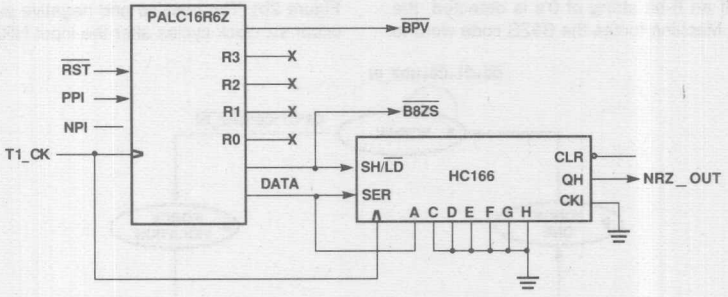


Figure 3. B8ZS Decoder

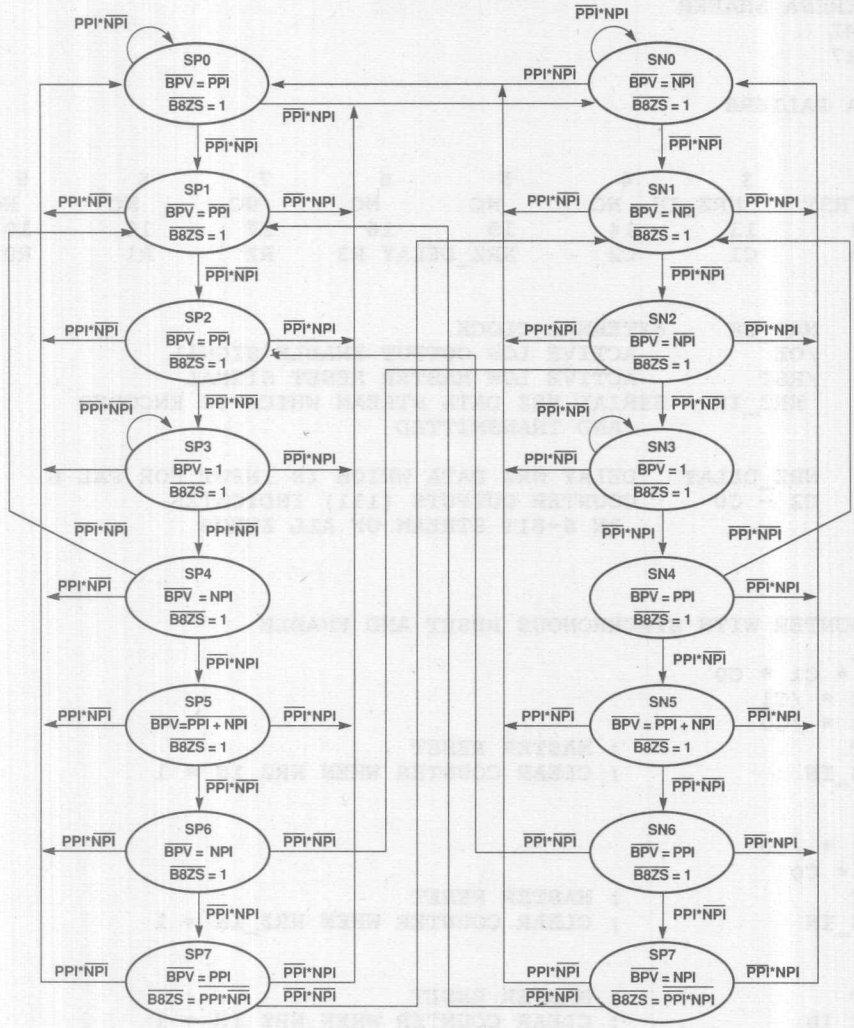
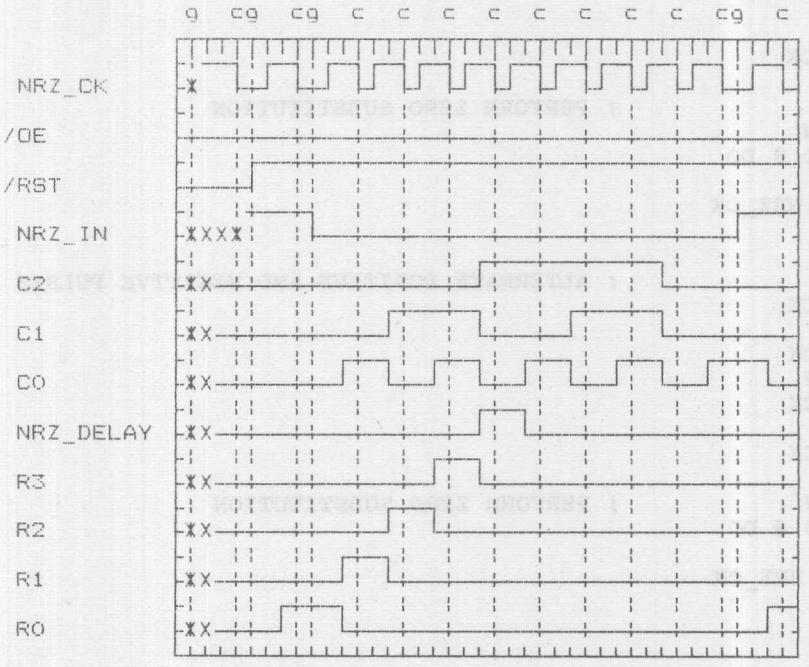


Figure 4. B8ZS Decoder State Machine

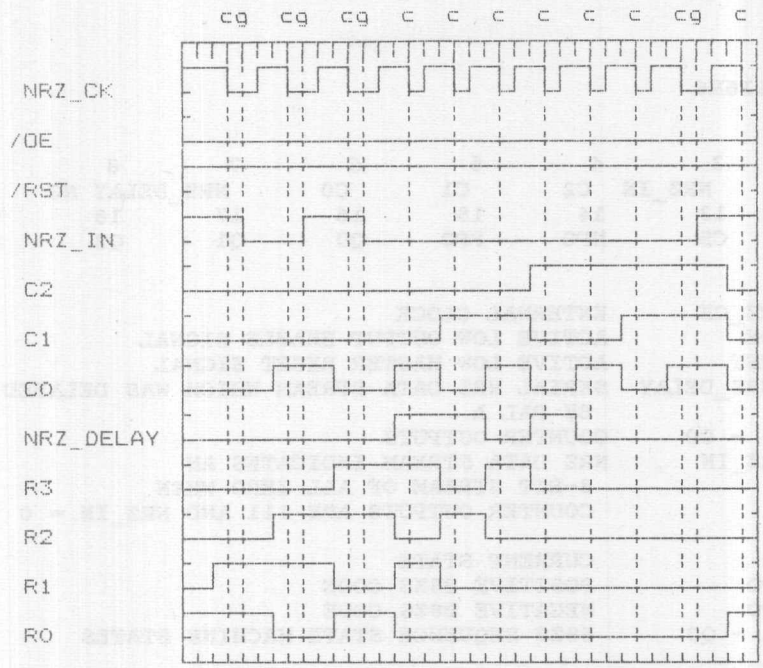
```
TITLE PAL_A
PATTERN B8ZS ENCODER PAL A
REVISION 1.02
AUTHOR THERESA SHAFER
COMPANY MMI
DATE 6/6/87

CHIP PAL_A PAL16R8

;PINS
;1       2       3       4       5       6       7       8       9
 NRZ_CK  /RST    NRZ_IN  NC      NC      NC      NC      NC      NC      GND
;11      12      13      14      15      16      17      18      19      20
/OE      C0      C1      C2      NRZ_DELAY R3    R2      R1      R0      VCC

;
; INPUTS:   NRZ_CK    EXTERNAL CLOCK
;           /OE         ACTIVE LOW OUTPUT ENABLE SIGNAL
;           /RST        ACTIVE LOW MASTER RESET SIGNAL
;            NRZ_IN   SERIAL NRZ DATA STREAM WHICH IS ENCODED
;                       AND TRANSMITTED

; OUTPUTS:  NRZ_DELAY   DELAY NRZ DATA WHICH IS INPUT FOR PAL B
;           C2 - C0     COUNTER OUTPUTS (111) INDICATES
;                       AN 8-BIT STREAM OF ALL ZEROS

EQUATIONS

; 3-BIT COUNTER WITH SYNCHRONOUS RESET AND ENABLE

/C2 := C2 * C1 * C0
     + /C2 * /C1
     + /C2 * /C0
     + RST                ; MASTER RESET
     + NRZ_IN             ; CLEAR COUNTER WHEN NRZ_IN = 1


/C1 := /C1 * /C0
     + C1 * C0
     + RST                ; MASTER RESET
     + NRZ_IN             ; CLEAR COUNTER WHEN NRZ_IN = 1

/C0 := C0
     + RST                ; MASTER RESET
     + NRZ_IN             ; CLEAR COUNTER WHEN NRZ_IN = 1

; 5-STAGE PIPELINE DELAY

/NRZ_DELAY := /R3 + RST

/R3 := /R2 + RST

/R2 := /R1 + RST

/R1 := /R0 + RST

/R0 := /NRZ_IN + RST
```

```
; ..........................................................
; ..........................................................

SIMULATION

TRACE_ON NRZ_CK /OE /RST NRZ_IN
         C2 C1 C0 NRZ_DELAY R3 R2 R1 R0

SETF OE                    ; ENABLE OUTPUT
     RST                   ; RESET
CLOCKF NRZ_CK

                           ; INITIALIZE
SETF NRZ_IN
     /RST
CLOCKF NRZ_CK

                           ; PERFORM ZERO SUBSTITUTION
SETF /NRZ_IN
FOR J:= 0 TO 8 DO
     BEGIN
     CLOCKF NRZ_CK
     END

SETF NRZ_IN                ; ALTERNATE POSITIVE AND NEGITIVE PULSES
CLOCKF NRZ_CK
SETF NRZ_IN
CLOCKF NRZ_CK
SETF /NRZ_IN
CLOCKF NRZ_CK
SETF NRZ_IN
CLOCKF NRZ_CK

SETF /NRZ_IN               ; PERFORM ZERO SUBSTITUTION
FOR J:= 0 TO 6 DO
     BEGIN
     CLOCKF NRZ_CK
     END

SETF NRZ_IN
CLOCKF NRZ_CK

TRACE_OFF
```

```
PALASM SIMULATION. V2.22 - MARKET RELEASE (11-19-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC. 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title    : PAL_A                        Author   : THERESA SHAFER
Pattern  : B8ZS ENCODER PAL A           Company  : MMI
Revision : 1.02                         Date     : 6/6/87


PAL_A
Page :  1
```



PAL_A
Page : 2


cg cg cg c c c c c c cg c
NRZ_CK
/OE
/RST
NRZ_IN
C2
C1
C0
NRZ_DELAY
R3
R2
R1
R0

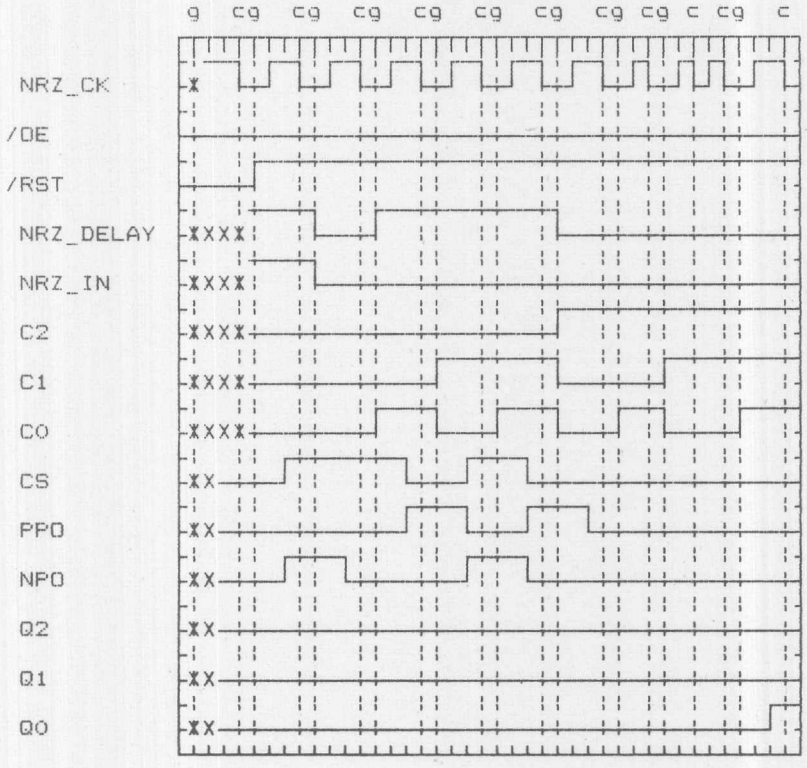
```
TITLE PAL_B
PATTERN B8ZS ENCODER PAL B
REVISION 1.04
AUTHOR THERESA SHAFER
COMPANY MMI
DATE 6/6/87

CHIP PAL_B PAL16R6

;PINS
;1      2       3       4       5       6       7       8       9
 NRZ_CK  /RST    NRZ_IN  C2      C1      C0      NRZ_DELAY NC    NC      GND
;11     12      13      14      15      16      17      18      19      20
/OE     NC      CS      NPO     PPO     Q0      Q1      Q2      NC      VCC

;
; INPUTS:   NRZ_CK      EXTERNAL CLOCK
;           /OE         ACTIVE LOW OUTPUT ENABLE SIGNAL
;           /RST        ACTIVE LOW MASTER RESET SIGNAL
;            NRZ_DELAY  SERIAL NRZ DATA STREAM WHICH WAS DELAYED
;                        BY PAL A
;           C2 - C0     COUNTER OUTPUTS
;           NRZ_IN      NRZ DATA STREAM INDICATES AN
;                        8-BIT STREAM OF ALL ZERO WHEN
;                        COUNTER OUTPUTS ARE 111 AND NRZ_IN = 0

; OUTPUTS:  CS          CURRENT STATE
;           PPO         POSITIVE B8ZS CODE
;           NPO         NEGATIVE B8ZS CODE
;           Q2 - Q0     B8ZS SEQUENCE STATE MACHINE STATES

EQUATIONS

; B8ZS ENCODING STATE MACHINE EQUATIONS
; STATE VARIATES = [CS,PPO,NPO]
; WITH THE FOLLOWING STATE ASSIGNMENT
;   P_ZERO   = [0,0,0]
;   NEGATIVE = [1,0,1]
;   N_ZERO   = [1,0,0]
;   POSITIVE = [0,1,0]

/CS := /CS * /NPO * /Q1 * Q0
     + CS * /PPO * Q1 * /Q0
     + /CS * /NPO * /NRZ_DELAY * /Q1
     + /CS * /NPO * Q2 * /Q1
     + CS * NRZ_DELAY * /PPO * /Q2 * /Q0
     + RST

/PPO := CS * /PPO * Q0 * /Q1
     + /CS * /NPO * /Q0
     + CS * /NRZ_DELAY * /PPO * /Q1
     + CS * /PPO * Q2 * /Q1
     + RST
```

```
/NPO := /CS * /NPO * /Q1 * Q0
            + /CS * /NPO * /NRZ_DELAY * /Q1
            + /CS * /NPO * Q2 * /Q1
            + CS * /PPO * /Q0
            + RST


; B8ZS SEQUENCE STATE MACHINE EQUATIONS
; STATE VARIATES [Q2,Q1,Q0]
; WITH THE FOLLOWING STATE ASSIGNMENTS
;    A = [0,0,0]
;    B = [0,0,1]
;    C = [0,1,0]
;    D = [1,0,0]
;    E = [1,0,1]
;    F = [1,1,0]

/Q2 := /Q2 * /Q1
     + Q2 * Q1
     + RST

/Q1 :=  /Q0
     + Q2 * Q1
     + RST

/Q0 := /Q2 * /Q1 * /Q0 * /C2
     + /Q2 * /Q1 * /Q0 * /C1
     + /Q2 * /Q1 * /Q0 * /C0
     + /Q2 * /Q1 * /Q0 * NRZ_IN
     + Q0
     + Q1
     + RST

; ..........................................................
; ..........................................................

SIMULATION

TRACE_ON NRZ_CK /OE /RST NRZ_DELAY NRZ_IN C2 C1 C0
         CS PPO NPO Q2 Q1 Q0

SETF OE                          ; ENABLE OUTPUT
     RST                         ; RESET
CLOCKF NRZ_CK

                                 ; INITIALIZE
SETF NRZ_DELAY
     /RST
     NRZ_IN /C2 /C1 /C0
CLOCKF NRZ_CK

                                 ; PERFORM NORMAL ENCODING
SETF /NRZ_DELAY
     /NRZ_IN /C2 /C1 /C0
CLOCKF NRZ_CK
SETF NRZ_DELAY
```

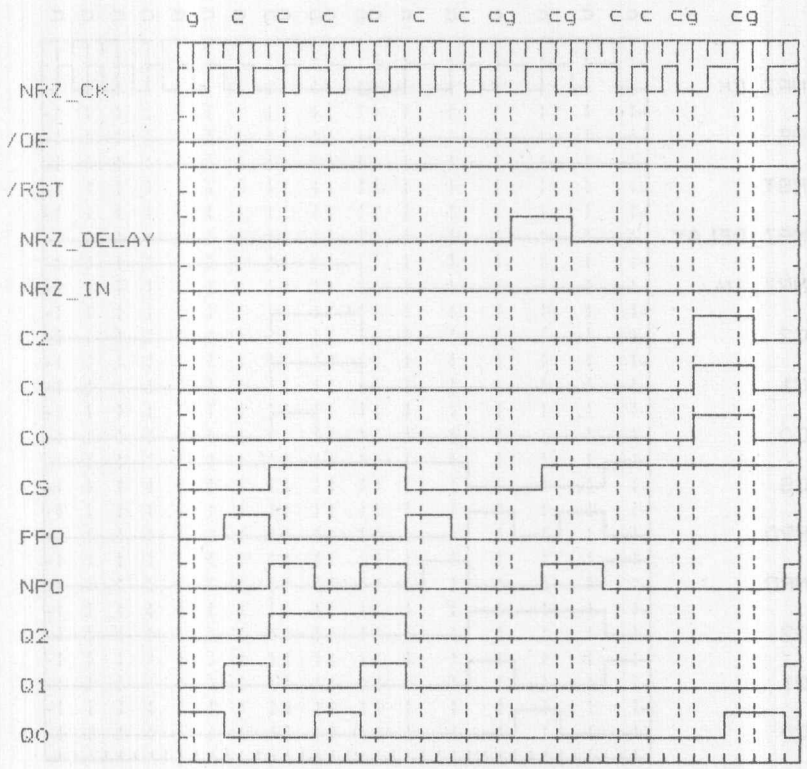
```
     /C2 /C1 C0
CLOCKF NRZ_CK
SETF NRZ_DELAY
     /C2 C1 /C0
CLOCKF NRZ_CK
SETF NRZ_DELAY
     /C2 C1 C0
CLOCKF NRZ_CK
SETF /NRZ_DELAY
     C2 /C1 /C0
CLOCKF NRZ_CK
SETF /NRZ_DELAY
     C2 /C1 C0
CLOCKF NRZ_CK
SETF /NRZ_DELAY
     C2 C1 /C0
CLOCKF NRZ_CK
SETF /NRZ_DELAY
     C2 C1 /C0
CLOCKF NRZ_CK
SETF C2 C1 C0
CLOCKF NRZ_CK
SETF /C2 /C1 /C0
CLOCKF NRZ_CK
FOR I:=0 TO 5 DO
     BEGIN
     CLOCKF NRZ_CK
     END

SETF /C2 /C1 /C0                    ; PERFORM B8ZS SUBSTITUTION
     NRZ_DELAY                      ; FORCE VIOLATION
CLOCKF NRZ_CK
SETF /NRZ_DELAY
CLOCKF NRZ_CK                       ; FORCE ONE
SETF /NRZ_DELAY
CLOCKF NRZ_CK                       ; FORCE ZERO
SETF /NRZ_DELAY
CLOCKF NRZ_CK                       ; FORCE VIOLATION
SETF C2 C1 C0
     /NRZ_DELAY
CLOCKF NRZ_CK                       ; FORCE ONE
SETF /C2 /C1 /C0
     /NRZ_DELAY
CLOCKF NRZ_CK
FOR I:=0 TO 5 DO
     BEGIN
     CLOCKF NRZ_CK                  ; PERFORM NORMAL ENCODING
     END

SETF /NRZ_DELAY
     NRZ_IN C2 C1 /C0
CLOCKF NRZ_CK
SETF C2 C1 C0
CLOCKF NRZ_CK
SETF /C2 /C1 /C0
```

```
CLOCKF NRZ_CK
FOR I:=0 TO 5 DO
     BEGIN
     CLOCKF NRZ_CK                ; PERFORM B8ZS SUBSTITUTION
     END

TRACE_OFF
```

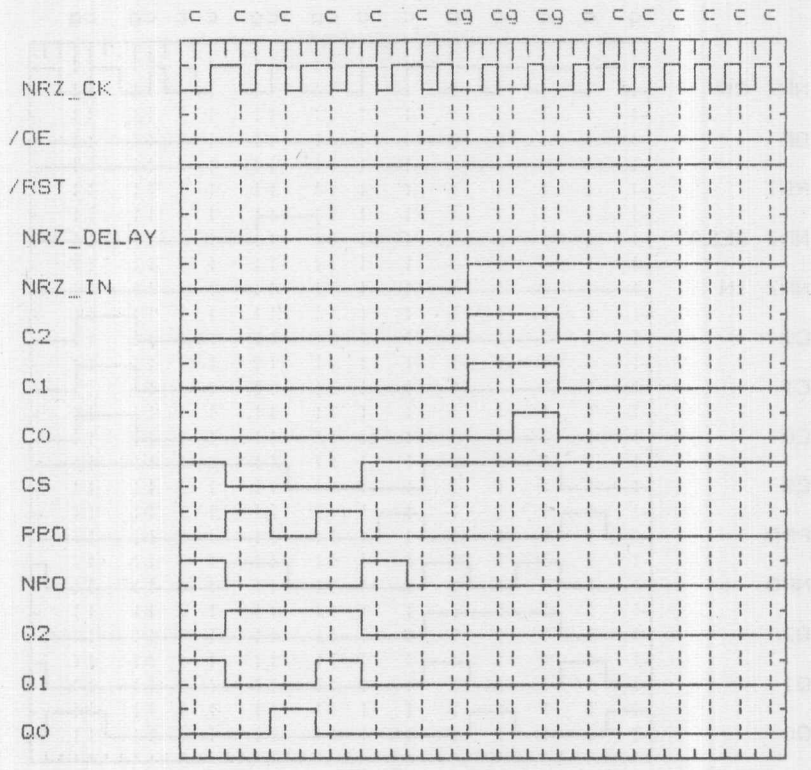
```
PALASM SIMULATION, V2.22 - MARKET RELEASE (11-19-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title    : PAL_B                    Author   : THERESA SHAFER
Pattern  : B8ZS ENCODER PAL B       Company  : MMI
Revision : 1.04                     Date     : 6/6/87


PAL_B
Page :  1
```



PAL_B
Page : 2


g c c c c c c cg cg c c cg cg
NRZ_CK
/OE
/RST
NRZ_DELAY
NRZ_IN
C2
C1
C0
CS
PP0
NP0
Q2
Q1
Q0

PAL_B
Page : 3


c c c c c c cg cg cg c c c c c c c
NRZ_CK
/OE
/RST
NRZ_DELAY
NRZ_IN
C2
C1
C0
CS
PPO
NPO
Q2
Q1
Q0

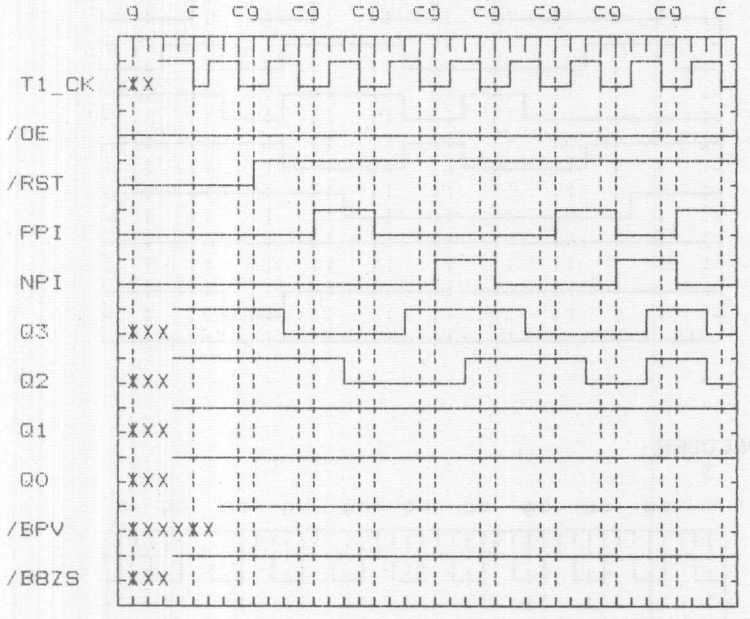
```
TITLE B8ZS_DECODER
PATTERN B8ZS DECODER PAL
REVISION 1.05
AUTHOR THERESA SHAFER
COMPANY MMI
DATE 6/6/87

CHIP B8ZS_DECODER PAL16R6

;PINS
;1       2       3       4       5       6       7       8       9
 T1_CK   /RST    PPI     NPI     NC      NC      NC      NC      NC      GND
;11      12      13      14      15      16      17      18      19      20
/OE      NC      Q0      Q1      Q2      Q3      /BPV    /B8ZS   DATA    VCC

;
; INPUTS:    T1_CK     EXTERNAL CLOCK
;            /OE          ACTIVE LOW OUTPUT ENABLE SIGNAL
;            /RST         ACTIVE LOW MASTER RESET SIGNAL
;             PPI         SERIAL POSITIVE PUSLE DATA STREAM
;                          TO BE DECODED
;             NPI         SERIAL NEGATIVE PUSLE DATA STREAM
;                          TO BE DECODED

;  OUTPUTS:    DATA       DATA  WHICH IS LOADED INTO  THE  SHIFT
;                          REGISTER
;           Q3 - Q0       STATE MACHINE OUTPUTS
;             /BPV        LOW ACTIVE SIGNAL WHICH INDICATES THAT
;                          BIPOLAR VIOLATION HAS OCCURRED
;             /B8ZS       LOW ACTIVE SIGNAL WHICH INDICATES THAT
;                          ZERO SUBSTITUTION HAS OCCURRED

EQUATIONS

; DATA OUTPUT

/DATA = /PPI * /NPI;

; /BPV AND /B8ZS FLAGS

BPV := PPI * /Q2 * Q1
     + PPI * /Q3 * Q2 * /Q0
     + NPI * Q2 * Q1
     + NPI * /Q3 * Q1 * /Q0
     + /PPI * /Q2 * /Q1 * Q0
     + /PPI * Q3 * /Q1 * Q0
     + /NPI * /Q3 * Q2 * /Q1
     + /NPI * Q2 * /Q1 * /Q0;

B8ZS := /PPI * NPI * /Q3 * Q2 * /Q1 * Q0
     + PPI * /NPI * Q3 * Q2 * /Q1 * Q0;

; STATE MACHINE WHICH DETECTS B8ZS SEQUENCE
; STATE VARIABLES [Q3,Q2,Q1,Q0]
; WITH THE FOLLOWING STATE ASSIGNMENTS
```

```
;       SP0 = [0,0,1,1]
;       SP1 = [1,0,1,1]
;       SP2 = [1,0,1,0]

;       SP3 = [0,0,0,0]
;       SP4 = [0,1,0,0]
;       SP5 = [0,0,1,0]
;       SP6 = [1,1,0,0]
;       SP7 = [1,1,0,1]
;       SN0 = [1,1,1,1]
;       SN1 = [0,1,1,1]
;       SN2 = [1,1,1,0]
;       SN3 = [1,0,0,0]
;       SN4 = [1,0,0,1]
;       SN5 = [0,1,1,0]
;       SN6 = [0,0,0,1]
;       SN7 = [0,1,0,1]


/Q3 := PPI * /RST
     + /NPI * /Q3 * /Q2 * /Q1 * /Q0 * /RST
     + /NPI * Q3 * /Q2 * Q1 * /Q0 * /RST
     + /NPI * /Q3 * Q2 * Q1 * /Q0 * /RST
     + /NPI * Q3 * /Q1 * Q0 * /RST
     + /NPI * Q3 * Q2 * Q0 * /RST
     + NPI * /Q3 * Q2 */Q1 * /Q0 * /RST;

/Q2 := PPI * Q1 * /RST
     + PPI * Q2 * /RST
     + /NPI * /Q3 * Q2 * /Q0 * /RST
     + /NPI * Q3 * Q1 * /Q0 * /RST
     + /PPI * /NPI * /Q3 * /Q1 * /RST
     + /NPI * /Q2 * Q1 * Q0 * /RST
     + Q3 * /Q2 * /Q1 * /Q0 * /RST
     + /Q3* Q2 * /Q1 * /Q0 * /RST;

/Q1 := PPI * /Q3 * /Q2 * /Q1 * /RST
     + /PPI * /NPI * /Q2 * /Q0 * /RST
     + /PPI * /NPI * Q1 * /Q0 * /RST
     + /PPI * NPI * Q3 * /Q1 * /Q0 * /RST;

/Q0 := PPI * /Q3 * /Q2 * /Q1 * /Q0 * /RST
     + PPI * Q3 * /Q2 * /Q1 * Q0 * /RST
     + /PPI * /NPI * /Q2 * /Q0 * /RST
     + /PPI * /NPI * Q3 * /Q0 * /RST
     + /PPI * /NPI * /Q3 * /Q2 * /Q1 * /RST
     + /PPI * /NPI * Q3 * /Q2 * Q1 * /RST
     + /PPI * /NPI * /Q3 * Q2 * Q1 * Q0 * /RST
     + /PPI * NPI * /Q3 * Q2 * /Q1 * /Q0 * /RST;

; ..........................................................
; ..........................................................
```

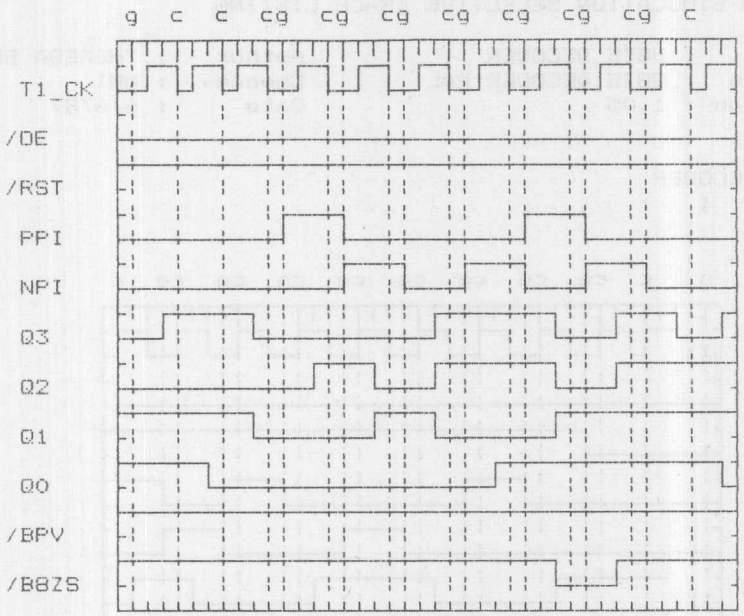
```
SIMULATION
TRACE_ON T1_CK /OE /RST PPI NPI
         Q3 Q2 Q1 Q0 /BPV /B8ZS

SETF OE                  ; ENABLE OUTPUT

     RST                 ; RESET
     /PPI /NPI
CLOCKF T1_CK
SETF RST
CLOCKF T1_CK
SETF /PPI /NPI
     /RST
CLOCKF T1_CK

                         ; NORMAL DECODING
SETF PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI

CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK

                         ; DETECT B8ZS SEQUENCE
                         ; + SEQUENCE
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK
                         ; SET-UP FOR NEXT SEQUENCE
SETF /PPI NPI
CLOCKF T1_CK
                         ; - SEQUENCE
SETF /PPI /NPI
CLOCKF T1_CK
```

```
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI
CLOCKF T1_CK
SETF PPI /NPI

CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI
CLOCKF T1_CK

                              ; DETECT BIPOLAR VIOLATIONS
SETF /PPI /NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF /PPI /NPI
CLOCKF T1_CK
SETF PPI /NPI
CLOCKF T1_CK

TRACE_OFF
```

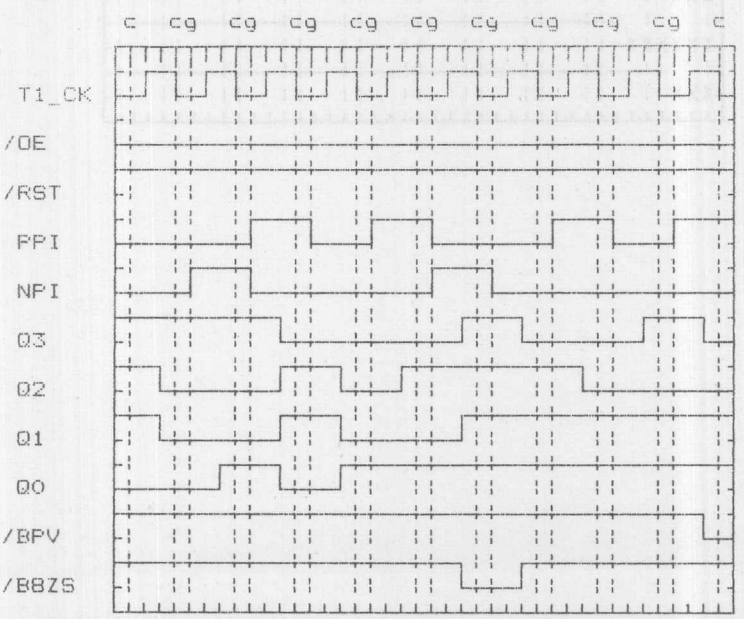
```
PALASM SIMULATION. V2.22 - MARKET RELEASE (11-19-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title     : B8ZS_DECODER                Author    : THERESA SHAFER
Pattern   : B8ZS DECODER PAL            Company   : MMI
Revision  : 1.05                        Date      : 6/6/87

B8ZS_DECODER
Page :  1
```



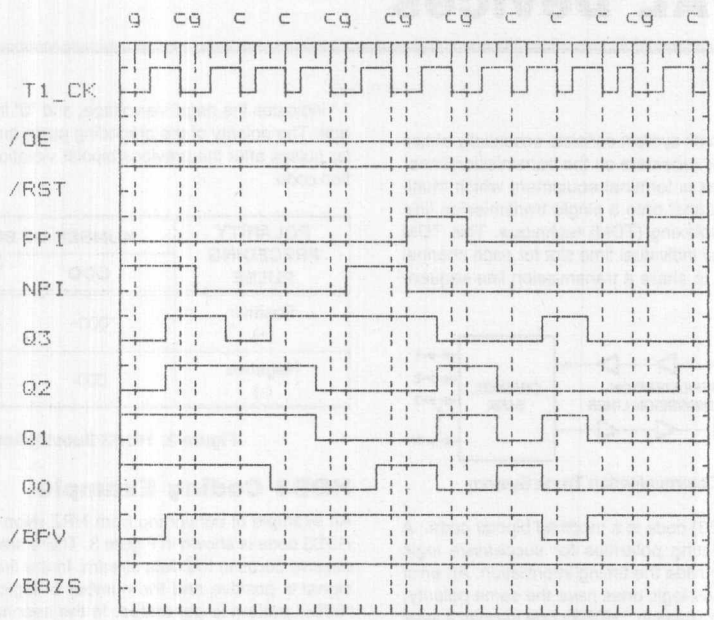
B8ZS_DECODER
Page : 2



B8ZS_DECODER
Page : 3



B8ZS_DECODER
Page : 4



HDB3 Line Coding Using PAL Devices

AN-176

Introduction

A digital communication trunk system consists essentially of two channel banks and several repeaters on the transmission lines (Figure 1). A channel bank is terminal equipment which multiplexes more than one channel onto a single transmission line using Time Division Multiplexing (TDM) technique. This TDM technique assigns an equal individual time slot for each channel to allow several channels to share a transmission line sequentially.

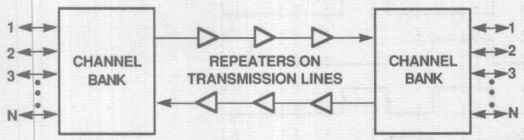


Figure 1. Digital Communication Trunk System

High Density Bipolar (HDB3) code is a modified bipolar code. A bipolar code uses alternating polarities for successive logic ones. The signal edges provide the timing information. An error occurs when two contiguous logic ones have the same polarity; this is defined as a "bipolar violation." HDB3 code permits a legal bipolar violation to prevent four or more contiguous zeros in the data stream. Three is the maximum number of contiguous zeros allowed before a violation is inserted to maintain timing information for a long string of logic zeros.

HDB3 Code

Code conversion rules for trunk lines are defined in the CCITT G703 recommendation, which includes two standards. One standard is T-carrier standard for North America and Japan; the other standard is CEPT standard for Europe and elsewhere. HDB3 code is recommended by CCITT for the CEPT standard, at the 2,048, 8,448, or 34,368 kbps rates.

Violation patterns used for four-zeros-substitution in HDB3 are shown in Figure 2. Four possible HDB3 substitution codes are: +00+, -00-, 000+, or 000-. The "+" indicates the positive voltage, "-" indicates the negative voltage, and "0" indicates the zero voltage. The polarity of the preceding pulse and the number of bipolar pulses after the previous bipolar violation select the substitution code.

POLARITY PRECEDING PULSE	NUMBER OF BIPOLAR PULSE	
	ODD	EVEN
Positive (+)	000+	-00-
Negative (-)	000-	+00+

Figure 2. HDB3 Substitution Table

HDB3 Coding Example

An example of converting from NRZ (Non-Return-Zero) data to HDB3 code is shown in Figure 3. There are four sets of four successive zeros in the data stream. In the first group the previous signal is positive and the number of logic ones is odd, so the "000+" pattern is generated. In the second group the previous signal is positive and the number of logic ones is even, so the "-00-" pattern is generated. In the third group the previous signal is negative and the number of logic ones is even, so the "+00+" pattern is generated . A "000-" pattern is generated in the fourth group because the previous signal is negative and the number of logic ones is odd.

HDB3 substitution codes force the fourth bit to be a bipolar violation. The violation is chosen by the following scheme, based on the number of bits since the previous violation:

(1) When the number is even, HDB3 substitution code forces the first zero to be a bipolar code (B) and the fourth zero to be a bipolar violation (V). Therefore, four successive zeros is converted to a bipolar signal, following two continuous zeros, and a bipolar violation with the first bipolar signal (B00V). The polarity of the bipolar violation (V) is the same as the polarity of the first bipolar signal (B).

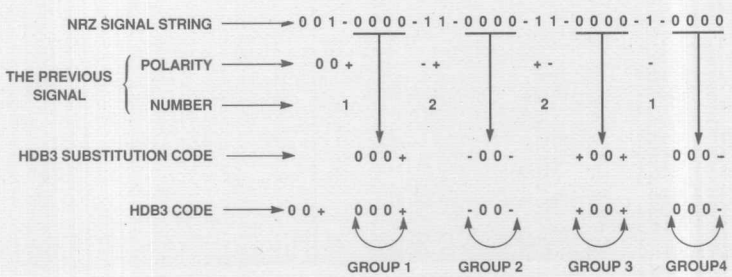


Figure 3. An Example of HDB3 Coding

(2) When the number is odd, HDB3 substitution code forces the fourth zero to be a bipolar violation (V). Therefore, HDB3 is converted as three continuous zeros followed by bipolar violation with the previous bipolar signal (000V). The polarity of the bipolar violation is the same as the polarity of the previous bipolar signal.

HDB3 Encoder Implementation

Figure 4 shows the block diagram of an HDB3 encoder, which includes three state machines and one 4-bit shift register. A DETECT_4_ZEROS state machine detects four successive zeros. An ODD_EVEN state machine checks the number of logic ones after the last bipolar violation. An HDB3 encoding state machine generates the HDB3 code and a RESET signal which clears the ODD_EVEN state machine. A 4-bit shift register delays the incoming NRZ data in order to synchronize with the 4-ZEROS signal from the DETECT_4_ZEROS state machine.

The HDB3 encoding state machine generates HDB3 code, depending on the DSIN, 4-ZEROS, ODD_EVEN, and POS_NEG signals. When DSIN is false and 4-ZEROS is false, the output is a zero. When DSIN is true and 4-ZEROS is false, the output generates a signal that alternates between the positive and negative signals. When DSIN is true and 4-ZEROS is true, an error flag is raised. HDB3 substitution code is generated when DSIN is false and 4-ZEROS is true. Four possible HDB3 substitution codes are generated, depending on POS_NEG and ODD_EVEN signals. These are summarized in Figure 5.

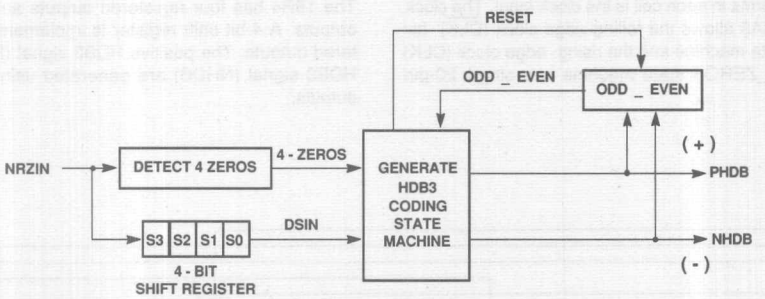


Figure 4. Block Diagram of an HDB3 Encoder

DSIN	4-ZEROS	POS_NEG	ODD_EVEN	CHARACTERISTICS
0	0	0	0	Zero output
0	0	0	1	
0	0	1	1	
0	0	1	0	
0	1	0	0	+00+
0	1	0	1	000-
0	1	1	1	000+
0	1	1	0	-00-
1	0	0	0	Alternating PHDB (+) and NHDB (-)
1	0	0	1	
1	0	1	1	
1	0	1	0	
1	1	0	0	Error
1	1	0	1	
1	1	1	1	
1	1	1	0	

Figure 5. HDB3 Encoding State Table

The three Programmable Array Logic (PAL) devices used to implement the entire HDB3 encoder are a 16RA8, a 20RS8, and a 16R4 (see Figure 6).

The 16RA8 includes two state machines: ODD_EVEN and DETECT_4_ZEROS. The ODD_EVEN's output signal is fed into the HDB3 decoding state machine. If the ODD_EVEN state machine is triggered by the same clock edge as the decoding state machine, the wrong result is generated because the encoding state machine always gets the ODD_EVEN's output signal one clock early. Therefore, the ODD_EVEN state machine must be triggered by the falling edge of the clock to compensate for this timing problem. However, the DETECT_4_ZEROS state machine is triggered by the rising edge of the clock. Both rising and falling edges are needed for these two state machines. The 16RA8 device offers the programmable clock feature because one of the product terms in each cell is the clock input. The clock flexibility of the 16RA8 allows the falling-edge clock ($\overline{CLK}$) for the ODD_EVEN state machine and the rising- edge clock (CLK) for the DETECT_4_ZEROS state machine in a single 20-pin PAL device.

The HDB3 encoding state machine diagram is shown in Figure 7. Four registers are used for fifteen states. Two of the four registers have ten product terms, one register has eight product terms, and the final register has seven product terms. Both the 20RS4 and 20RS8 have the product-sharing feature. The 20RS8 is used to implement this HDB3 encoding state machine because the product sharing feature is sixteen product terms shared by two registers exclusively, and in this case at least seventeen product terms are required per register pair. The design fits in the 20RS8 by using the product terms from two registers for one output, and leaving the other output as a no-connect. In addition to the HDB3 encoding state machine, a reset signal and an error message are generated in this 20RS8. The reset signal clears the ODD_EVEN state machine. The error message occurs when DSIN and 4_ZEROS both are high simultaneously.

The 16R4 has four registered outputs and four combinatorial outputs. A 4-bit shift register is implemented using four registered outputs. The positive HDB3 signal (PHDB) and negative HDB3 signal (NHDB) are generated using two combinatorial outputs.

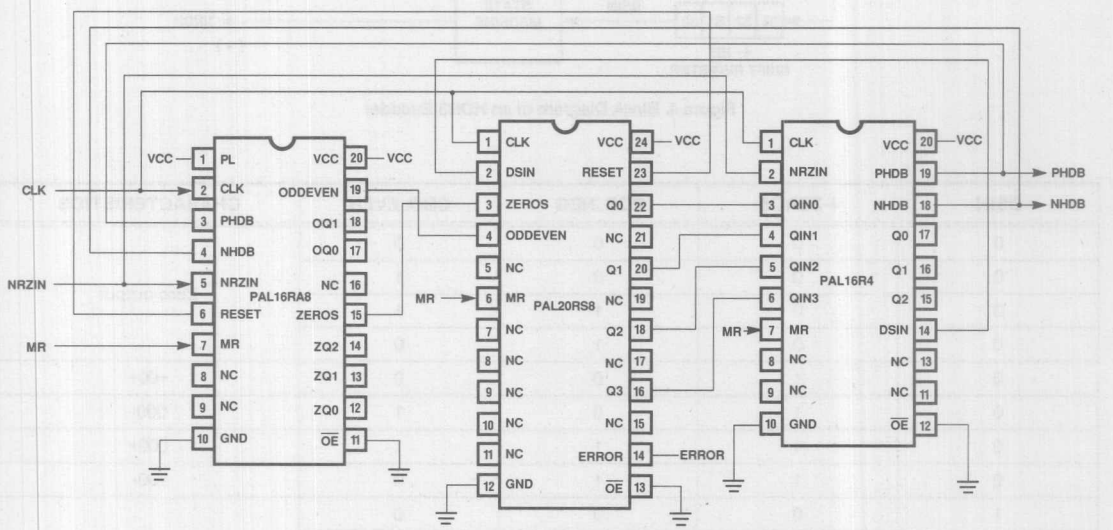
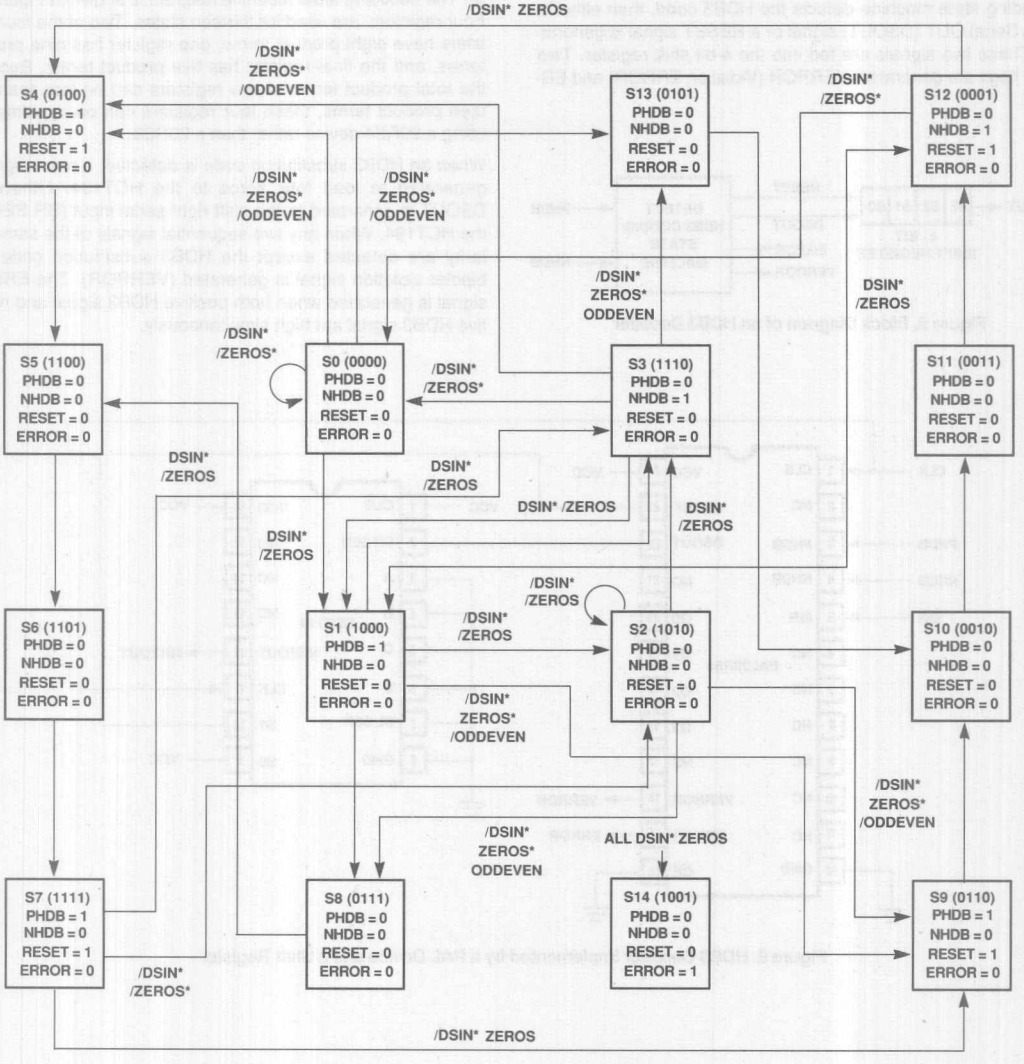


Figure 6. HDB3 Encoder Implemented by Three PAL Devices



Notes:

1. Inputs: DSIN, ZEROS, and ODDEVEN.
2. State Outputs: PHDB, NHDB, RESET and ERROR.

Figure 7. HDB3 Encoding State Machine

HDB3 Decoder Implementation

The block diagram of the HDB3 decoder is shown in Figure 8. A decoding state machine detects the HDB3 code, then either a Data Serial OUT (DSOUT) signal or a RESET signal is generated. These two signals are fed into the 4-bit shift register. Two error flags are generated: VERROR (Violation ERROR) and ERROR.

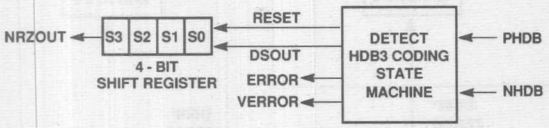


Figure 8. Block Diagram of an HDB3 Decoder

A single 24-pin PAL device (20RS4) and a 4-bit shift register (HCT194) are used to implement the HDB3 decoder (see Figure 9). The decoding state machine diagram is shown in Figure 10. Four registers are used for thirteen states. Two of the four registers have eight product terms, one register has nine product terms, and the final register has five product terms. Because the total product terms of two registers can be less than sixteen product terms, these four registers can be implemented using a 20RS4 device rather than a 20RS8.

When an HDB3 substitution code is detected, a reset signal is generated to load four zeros to the HCT194. Otherwise, DSOUT is generated to the shift right serial input (SR SER) of the HCT194. When any two sequential signals of the same polarity are detected except the HDB3 substitution code, the bipolar violation signal is generated (VERROR). The ERROR signal is generated when both positive HDB3 signal and negative HDB3 signal are high simultaneously.

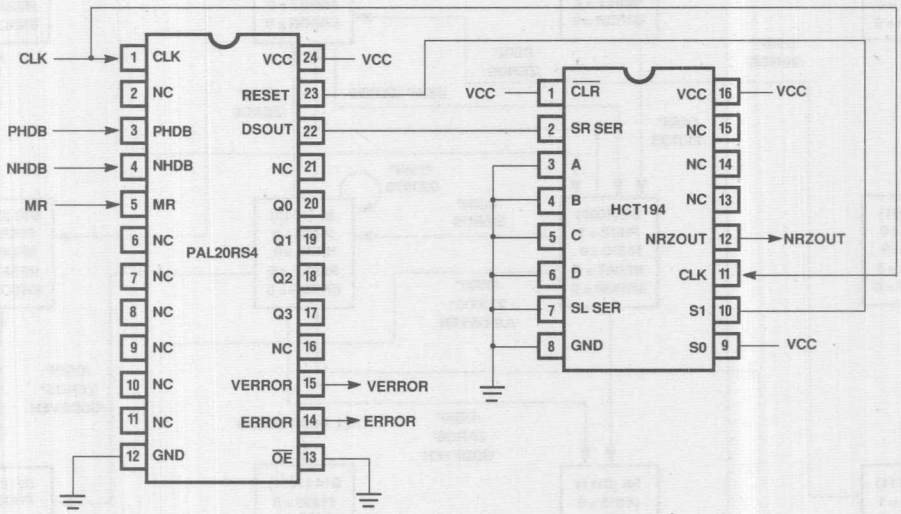
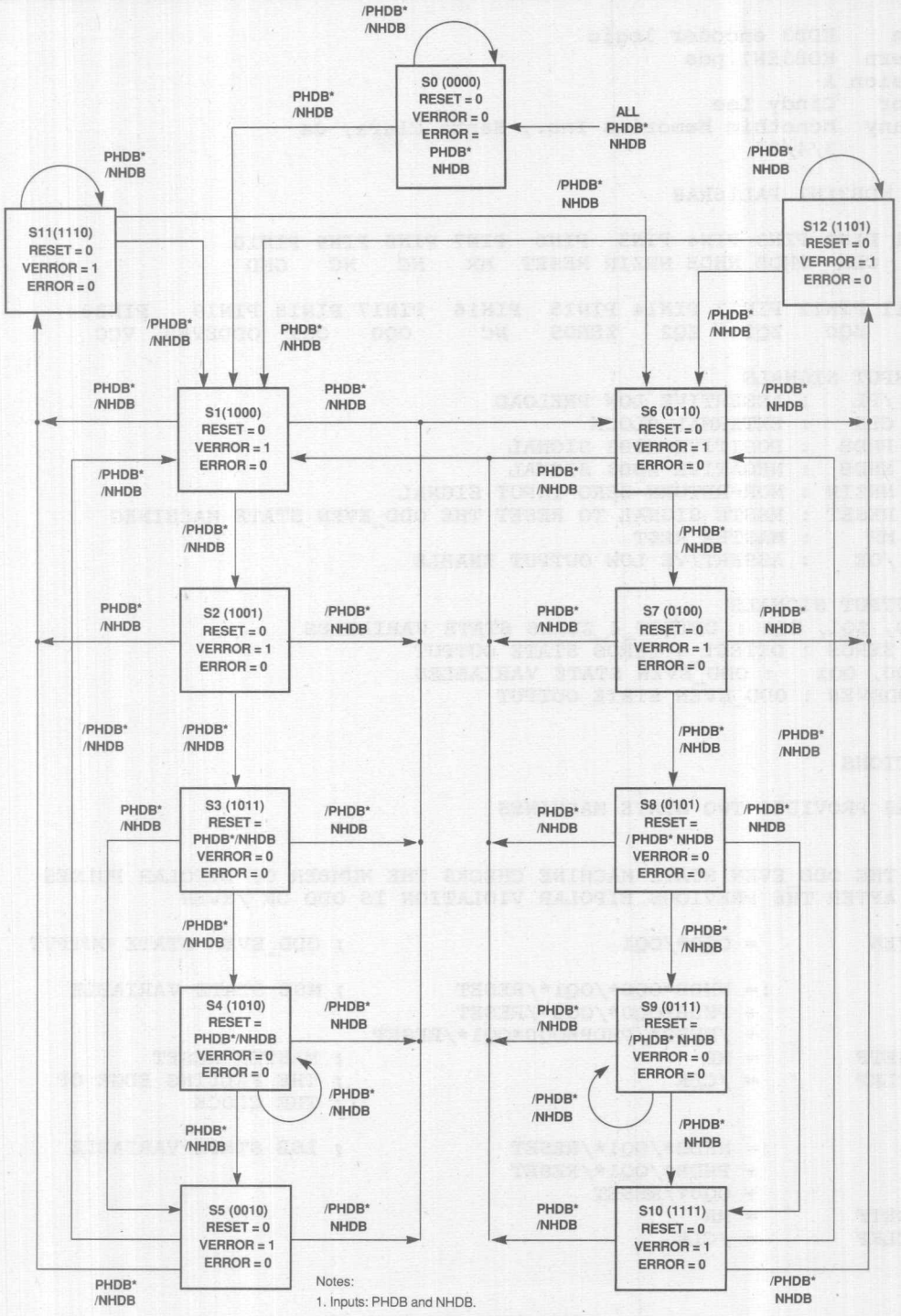


Figure 9. HDB3 Decoder Implemented by a PAL Device and a Shift Register



Notes:
1. Inputs: PHDB and NHDB.
2. State Outputs: RESET, VERROR and ERROR.

```
Title    HDB3 encoder logic
Pattern  HDB3EN1.pds
Revision A
Author   Cindy Lee
Company  Monothic Memories Inc., Santa Clara, Ca
Date     3/4/87

CHIP HDB3EN1 PAL16RA8

;PIN1 PIN2 PIN3 PIN4 PIN5  PIN6  PIN7 PIN8 PIN9 PIN10
/PL   CLK  PHDB NHDB NRZIN RESET  MR   NC   NC   GND

;PIN11 PIN12 PIN13 PIN14 PIN15  PIN16  PIN17 PIN18 PIN19   PIN20
/OE    ZQ0   ZQ1   ZQ2   ZEROS   NC     OQ0   OQ1  ODDEVEN  VCC

;  INPUT SIGNALS
;    /PL    : ASSERTIVE LOW PRELOAD
;    CLK    : EXTERNAL CLOCK
;    PHDB   : POSITIVE HDB3 SIGNAL
;    NHDB   : NEGATIVE HDB3 SIGNAL
;    NRZIN  : NOR-RETURN-ZERO INPUT SIGNAL
;    RESET  : RESTE SIGNAL TO RESET THE ODD_EVEN STATE MACHINEG
;    MR     : MASTER REST
;    /OE    : ASSERTIVE LOW OUTPUT ENABLE

;  OUTPUT SIGNALS
; ZQ0, ZQ1, ZQ2 : DETECT_4_ZEROS STATE VARIABLES
;    ZEROS : DETECT_4_ZEROS STATE OUTPUT
;  OQ0, OQ1   : ODD_EVEN STATE VARIABLES
;  ODDEVEN : ODD_EVEN STATE OUTPUT


EQUATIONS

;16RA8 PROVIDES TWO STATE MACHINES


;(1) THE ODD_EVEN STATE MACHINE CHECKS THE NUMBER OF BIPOLAR PULSES
;    AFTER THE PREVIOUS BIPOLAR VIOLATION IS ODD OR /EVEN

ODDEVEN        = OQ0*/OQ1                        ; ODD_EVEN STATE OUTPUT

OQ1           := NHDB*OQ0*/OQ1*/RESET            ; MSB STATE VARIABLE
               + PHDB*OQ0*/OQ1*/RESET
               + /NHDB*/PHDB*OQ0*OQ1*/RESET
OQ1.SETF       = MR                              ; MASTER RESET
OQ1.CLKF       = /CLK                            ; THE FALLING EDGE OF
                                                 ; THE CLOCK

OQ0           := NHDB*/OQ1*/RESET                ; LSB STATE VARIABLE
               + PHDB*/OQ1*/RESET
               + OQ0*/RESET
OQ0.SETF       = MR
OQ0.CLKF       = /CLK
```

```
;(2) THE DETECT_4_ZEROS STATE MACHINE DETECTS FOUR CONTINUOUS ZEROS

ZEROS          = /ZQ0 * /ZQ1 * ZQ2              ; STATE OUTPUT

ZQ2           := /NRZIN * /ZQ0 * ZQ1 * /ZQ2     ; MSB STATE VARIABLE

ZQ2.SETF       = MR                             ; MASTER RESET
ZQ2.CLKF       = CLK                            ; EXTERNAL CLOCK

ZQ1           := /NRZIN * ZQ0 * /ZQ2
ZQ1.SETF       = MR
ZQ1.CLKF       = CLK

ZQ0           := /NRZIN * /ZQ0 * /ZQ1           ; LSB STATE VARIABLE
               + /NRZIN * /ZQ1 * /ZQ2
ZQ0.SETF       = MR
ZQ0.CLKF       = CLK


SIMULATION

TRACE_ON CLK MR NRZIN PHDB NHDB RESET    ODDEVEN  ZEROS

SETF /CLK  MR /NRZIN /PHDB /NHDB /RESET OE /PL  ;RESET CONDITION
SETF  CLK /MR                                    ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /MR /NRZIN /PHDB /NHDB /RESET  ;VECTOR 1, STATE 0
SETF  CLK                                ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN /PHDB /NHDB /RESET      ;VECTOR 2, STATE 0
SETF  CLK                                ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN /PHDB /NHDB /RESET      ;VECTOR 3, STATE 0
SETF  CLK                                ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN /PHDB /NHDB /RESET      ;VECTOR 4, STATE 0
SETF  CLK                                ;/DSIN, ZEROS, /ODDEVEN

SETF /CLK  NRZIN  PHDB /NHDB  RESET      ;VECTOR 5, STATE 4
SETF  CLK                                ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK  NRZIN /PHDB /NHDB /RESET      ;VECTOR 6, STATE 5
SETF  CLK                                ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK  NRZIN /PHDB /NHDB /RESET      ;VECTOR 7, STATE 6
SETF  CLK                                ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN  PHDB /NHDB  RESET      ;VECTOR 8, STATE 7
SETF  CLK                                ;DSIN, /ZEROS, /ODDEVEN

SETF /CLK  NRZIN /PHDB  NHDB /RESET      ;VECTOR 9, STATE 3
SETF  CLK                                ;DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN  PHDB /NHDB /RESET      ;VECTOR 10, STATE 1
```

```
SETF  CLK                                    ;DSIN, /ZEROS, /ODDEVEN

SETF /CLK  NRZIN /PHDB  NHDB /RESET          ;VECTOR 11, STATE 3
SETF  CLK                                    ;/DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN /PHDB /NHDB /RESET          ;VECTOR 12, STATE 0
SETF  CLK                                    ;DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN  PHDB /NHDB /RESET          ;VECTOR 13, STATE 1
SETF  CLK                                    ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN /PHDB /NHDB /RESET          ;VECTOR 14, STATE 2
SETF  CLK                                    ;DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN /PHDB  NHDB /RESET          ;VECTOR 15, STATE 3
SETF  CLK                                    ;/DSIN, ZEROS, ODDEVEN

SETF /CLK  NRZIN /PHDB /NHDB /RESET          ;VECTOR 16, STATE 13
SETF  CLK                                    ;/DSIN, /ZEROS, ODDEVEN

SETF /CLK  NRZIN /PHDB /NHDB /RESET          ;VECTOR 17, STATE 10
SETF  CLK                                    ;/DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN /PHDB /NHDB /RESET          ;VECTOR 18, STATE 11
SETF  CLK                                    ;/DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN /PHDB  NHDB  RESET          ;VECTOR 19, STATE 12
SETF  CLK                                    ;DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN  PHDB /NHDB /RESET          ;VECTOR 20, STATE 1
SETF  CLK                                    ;DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN /PHDB  NHDB /RESET          ;VECTOR 21, STATE 3
SETF  CLK                                    ;/DSIN, ZEROS, /ODDEVEN

SETF /CLK  NRZIN  PHDB /NHDB  RESET          ;VECTOR 22, STATE 4
SETF  CLK                                    ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN /PHDB /NHDB /RESET          ;VECTOR 23, STATE 5
SETF  CLK                                    ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK  NRZIN /PHDB /NHDB /RESET          ;VECTOR 24, STATE 6
SETF  CLK                                    ;/DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN  PHDB /NHDB  RESET          ;VECTOR 25, STATE 7
SETF  CLK                                    ;DSIN, /ZEROS, /ODDEVEN

SETF /CLK  NRZIN /PHDB  NHDB /RESET          ;VECTOR 26, STATE 3
SETF  CLK                                    ;/DSIN, /ZEROS, ODDEVEN

SETF /CLK  NRZIN /PHDB /NHDB /RESET          ;VECTOR 27, STATE 0
SETF  CLK                                    ;DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN  PHDB /NHDB /RESET          ;VECTOR 28, STATE 1
SETF  CLK                                    ;/DSIN, /ZEROS, /ODDEVEN
```

```
SETF /CLK /NRZIN /PHDB /NHDB /RESET      ;VECTOR 29, STATE 2
SETF  CLK                                ;DSIN, /ZEROS, /ODDEVEN

SETF /CLK /NRZIN /PHDB  NHDB /RESET      ;VECTOR 30, STATE 3
SETF  CLK                                ;DSIN, /ZEROS, ODDEVEN

SETF /CLK /NRZIN  PHDB /NHDB /RESET      ;VECTOR 31, STATE 1
SETF  CLK                                ;/DSIN, ZEROS, /ODDEVEN

TRACE_OFF
```

```
Title    HDB3 encoder
Pattern  HDB3EN2.pds
Revision A
Author   Cindy Lee
Company  Monolithic Memories Inc., Santa Clara, Ca
Date     3/9/87

CHIP HDB3EN2 PAL20RS8

;PIN1 PIN2 PIN3   PIN4   PIN5 PIN6
CLK   DSIN ZEROS ODDEVEN  NC   MR

;PIN7 PIN8 PIN9 PIN10 PIN11 PIN12
NC    NC   NC   NC    NC    GND

;PIN13 PIN14 PIN15 PIN16 PIN17 PIN18
/OE    ERROR  NC    Q3    NC    Q2

;PIN19 PIN20 PIN21 PIN22 PIN23 PIN24
NC     Q1    NC    Q0    RESET  VCC

;  INPUT SIGNALS
;    CLK     : EXTERNAL CLOCK
;    DSIN    : SERIAL DATA INPUT SIGNAL
;    ZEROS   : FOUR ZEROS DETECTION FLAG INPUT
;    ODDEVEN : ODD OR /EVEN SIGNAL
;    MR      : MASTER RESET
;    /OE     : ASSERTIVE LOW OUTPUT ENABLE

;  OUTPUT SIGNALS
;    ERROR   : ERROR FLAG
; Q3, Q2, Q1, Q0  : HDB3 DECODING STATE MACHINE STATE VARIABLES
;    RESET   : RESET SIGNAL


EQUATIONS

;20RS8 GENERATES TWO SIGNALS AND ONE STATE MACHINE

;(1) A RESET SIGNAL CLEARS THE ODD_EVEN STATE MACHINE

RESET   = /Q0 *  Q2 * /Q3                ; STATE 4 AND STATE 9
        +  Q0 * /Q1 * /Q2 * /Q3          ; STATE 12
        +  Q0 *  Q1 *  Q2 *  Q3          ; STATE 7


;(2) A ERROR SIGNAL IS RAISED WHEN DSIN AND 4-ZEROS BOTH ARE
;    HIGH AT THE SAME TIME

ERROR   =  Q0 * /Q1 * /Q2 *  Q3          ; STATE 14


;(3) HDB3 ENCODING STATES MACHINE

Q3     := /Q0 * /Q1 * Q2 * /MR           ; MSB STATE VARIABLE
```

```
        + /Q0 * /Q2 * Q3 * /ZEROS * /MR
        + DSIN * /Q0 * ZEROS * /MR
        + DSIN * Q2 * Q3 * /MR
        + DSIN * /Q1 * /Q2 * /Q3 * /MR
        + DSIN * /Q1 * ZEROS * /MR
        + /Q1 * Q2 * Q3 * /MR
        + DSIN * /Q3 * ZEROS * /MR
        + Q0 * Q1 * Q2 * /Q3 * /MR
        + Q0 * Q2 * Q3 * /ZEROS * /MR

Q2      := /DSIN * /Q0 * /Q1 * Q2 * /MR
        + /DSIN * Q0 * Q1 * Q2 * /Q3 * /MR
        + /Q0 * /Q1 * Q2 * /ZEROS * /MR
        + /Q1 * Q2 * Q3 * /ZEROS * /MR
        + Q0 * Q1 * Q2 * /Q3 * /ZEROS * /MR
        + DSIN * /Q0 * /Q2 * Q3 * /ZEROS * /MR
        + DSIN * Q0 * Q2 * Q3 * /ZEROS * /MR
        + /DSIN * /Q0 * Q3 * ZEROS * /MR
        + /DSIN * /Q1 * /Q2 * /Q3 * ZEROS * /MR
        + /DSIN * Q2 * Q3 * ZEROS * /MR

Q1      := /DSIN * /Q0 * Q1 * /Q3 * /MR
        + /DSIN * Q0 * /Q1 * Q2 * /MR
        + /Q0 * Q1 * /Q3 * /ZEROS * /MR
        + Q0 * /Q1 * Q2 * /ZEROS * /MR
        + /Q0 * /Q2 * Q3 * /ZEROS * /MR
        + Q0 * Q2 * Q3 * /ZEROS * /MR
        + /DSIN * /Q0 * /Q2 * Q3 * /MR
        + /DSIN * Q0 * Q2 * Q3 * /MR

Q0      := DSIN * /Q0 * ZEROS * /MR       ; LSB STATE VARIABLE
        + DSIN * /Q1 * ZEROS * /MR
        + /Q1 * Q2 * Q3 * /MR
        + Q1 * /Q2 * /Q3 * /MR
        + DSIN * Q2 * ZEROS * /MR
        + ODDEVEN * /Q0 * Q3 * ZEROS * /MR
        + ODDEVEN * /Q0 * /Q2 * ZEROS * /MR

SIMULATION

TRACE_ON MR CLK DSIN ZEROS ODDEVEN Q3 Q2 Q1 Q0 RESET ERROR

SETF OE MR DSIN  ZEROS  ODDEVEN   ;RESET CONDITION
CLOCKF CLK

SETF /MR

SETF  DSIN   ZEROS  /ODDEVEN      ;VECTOR 1, STATE 14, 1001
CLOCKF CLK                        ;ERROR

SETF /DSIN  /ZEROS  /ODDEVEN      ;VECTOR 2, STATE 0, 0000
CLOCKF CLK

SETF  DSIN  /ZEROS  /ODDEVEN      ;VECTOR 3, STATE 1, 1000
CLOCKF CLK                        ;PHDB
```

```
SETF /DSIN   ZEROS   ODDEVEN      ;VECTOR 4, STATE 8, 0111
CLOCKF CLK

SETF /DSIN  /ZEROS   ODDEVEN      ;VECTOR 5, STATE 5, 1100
CLOCKF CLK

SETF /DSIN  /ZEROS   ODDEVEN      ;VECTOR 6, STATE 6, 1101
CLOCKF CLK

SETF /DSIN  /ZEROS   ODDEVEN      ;VECTOR 7, STATE 7, 1111
CLOCKF CLK                        ;PHDB, RESET

SETF  DSIN  /ZEROS  /ODDEVEN      ;VECTOR 8, STATE 3, 1110
CLOCKF CLK                        ;NHDB

SETF  DSIN  /ZEROS   ODDEVEN      ;VECTOR 9, STATE 1, 1000
CLOCKF CLK                        ;PHDB

SETF /DSIN   ZEROS  /ODDEVEN      ;VECTOR 10, STATE 9, 0110
CLOCKF CLK                        ;NHDB, RESET

SETF /DSIN  /ZEROS  /ODDEVEN      ;VECTOR 11, STATE 10, 0010
CLOCKF CLK

SETF /DSIN  /ZEROS  /ODDEVEN      ;VECTOR 12, STATE 11, 0011
CLOCKF CLK

SETF /DSIN  /ZEROS  /ODDEVEN      ;VECTOR 13, STATE 12, 0001
CLOCKF CLK                        ;NHDB, RESET

SETF  DSIN  /ZEROS  /ODDEVEN      ;VECTOR 14, STATE 1, 1000
CLOCKF CLK                        ;PHDB

SETF  DSIN  /ZEROS   ODDEVEN      ;VECTOR 15, STATE 3, 1110
CLOCKF CLK                        ;NHDB

SETF /DSIN   ZEROS  /ODDEVEN      ;VECTOR 16, STATE 4, 0100
CLOCKF CLK                        ;PHDB, RESET

SETF /DSIN  /ZEROS  /ODDEVEN      ;VECTOR 17, STATE 5, 1100
CLOCKF CLK

SETF /DSIN  /ZEROS  /ODDEVEN      ;VECTOR 18, STATE 6, 1101
CLOCKF CLK

SETF /DSIN  /ZEROS  /ODDEVEN      ;VECTOR 19, STATE 7, 1111
CLOCKF CLK                        ;PHDB, RESET

SETF  DSIN  /ZEROS  /ODDEVEN      ;VECTOR 20, STATE 3, 1110
CLOCKF CLK                        ;NHDB

SETF /DSIN   ZEROS   ODDEVEN      ;VECTOR 21, STATE 13, 0101
CLOCKF CLK

SETF /DSIN  /ZEROS   ODDEVEN      ;VECTOR 22, STATE 10, 0010
CLOCKF CLK
```

```
SETF /DSIN  /ZEROS   ODDEVEN      ;VECTOR 23, STATE 11, 0011
CLOCKF CLK

SETF /DSIN  /ZEROS   ODDEVEN      ;VECTOR 24, STATE 12, 0001
CLOCKF CLK                        ;NHDB, RESET

TRACE_OFF

  PALASM SIMULATION, V2.22 - MARKET RELEASE (11-19-86)
  (C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
  PALASM SIMULATION SELECTIVE TRACE LISTING
```

```
Title    HDB3 shift register
Pattern  HDB3EN3.pds
Revision A
Author   Cindy Lee
Company  Monothic Memories Inc., Santa Clara, Ca
Date     2/24/87

CHIP HDB3EN3 PAL16R4

;PIN1 PIN2  PIN3 PIN4 PIN5 PIN6 PIN7 PIN8 PIN9 PIN10
CLK   NRZIN QIN0 QIN1 QIN2 QIN3  MR   NC   NC  GND

;PIN11 PIN12 PIN13 PIN14 PIN15 PIN16 PIN17 PIN18 PIN19 PIN20
/OE     NC    NC   DSIN   Q2    Q1    Q0   NHDB  PHDB  VCC

;  INPUT SIGNALS
;    CLK   : EXTERNAL CLOCK
;    NRZIN : NON-RETURN-ZERO INPUT SIGNAL
; QIN0, QIN1, QIN2, QIN3 : FOUR STATE VARIABLE INPUTS
;    MR    : MASTER RESET
;    /OE   : ASSERTIVE LOW OUTPUT ENABLE

;  OUTPUT SIGNALS
; DSIN, Q2, Q1, Q0 : 4-BIT SHIFT REGISTER OUTPUT SIGNALS
;    PHDB  : POSITIVE HDB3 SIGNAL
;    NHDB  : NEGATIVE HDB3 SIGNAL


EQUATIONS

;16R4 GENERATES TWO HDB SIGNALS AND PROVIDES A 4-BIT SHIFT REGISTER

;(1) A POSITIVE HDB3 AND A NEGATIVE HDB3 SIGNALS ARE GENERATED

/PHDB   = QIN0 * /QIN1                ; THE PHDB IS GENERATED AT
        + /QIN0 * QIN2 * QIN3         ; STATE 1, STATE 4, OR STATE 7
        + QIN1 * QIN2 * /QIN3         ; THIS EQUATION IS THE INVERTING
        + QIN1 * /QIN2                ; LOGIC OF THE PHDB SIGNAL
        + /QIN2 * /QIN3

/NHDB   = /QIN0 * /QIN1               ; THE NHDB IS GENERATED AT
        +  QIN1 * /QIN2               ; STATE 3, STATE 9, OR ATATE 12
        + /QIN2 * QIN3                ; THIS EQUATION IS THE INVERTING
        + QIN0 * QIN2                 ; LOGIC OF THE NHDB SIGNAL


;(2) THIS IS A 4-BIT SHIFT REGISTER WITH MASTER RESET

/Q0    := /NRZIN                      ; LSB OF SHIFT REGISTER
        +  MR                         ; MASTER RESET

/Q1    := /Q0
        +  MR

/Q2    := /Q1
```

```
        +  MR

/DSIN   := /Q2                          ; MSB OF SHIFT REGISTER
        +  MR


SIMULATION

TRACE_ON MR CLK QIN0 QIN1 QIN2 QIN3 PHDB NHDB NRZIN DSIN

SETF  OE MR  QIN0 /QIN1 /QIN2  QIN3  NRZIN      ;RESET CONDITION
CLOCKF CLK

SETF /MR  QIN0 /QIN1 /QIN2  QIN3  NRZIN         ;VECTOR 1
CLOCKF CLK

SETF /MR  QIN0 /QIN1 /QIN2  QIN3  NRZIN         ;VECTOR 2
CLOCKF CLK

SETF /MR /QIN0 /QIN1 /QIN2 /QIN3 /NRZIN         ;VECTOR 3
CLOCKF CLK

SETF /MR /QIN0 /QIN1 /QIN2  QIN3  NRZIN         ;VECTOR 4, PHDB
CLOCKF CLK                                      ;DSIN STARTS

SETF /MR /QIN0  QIN1 /QIN2  QIN3 /NRZIN         ;VECTOR 5
CLOCKF CLK                                      ;DSIN

SETF /MR /QIN0 /QIN1  QIN2  QIN3 /NRZIN         ;VECTOR 6
CLOCKF CLK                                      ;/DSIN

SETF /MR  QIN0 /QIN1  QIN2  QIN3  NRZIN         ;VECTOR 7
CLOCKF CLK                                      ;DSIN

SETF /MR  QIN0  QIN1  QIN2  QIN3  NRZIN         ;VECTOR 8, PHDB
CLOCKF CLK                                      ;/DSIN

SETF /MR  QIN0 /QIN1  QIN2 /QIN3 /NRZIN         ;VECTOR 9
CLOCKF CLK                                      ;/DSIN

SETF /MR /QIN0  QIN1 /QIN2 /QIN3  NRZIN         ;VECTOR 10
CLOCKF CLK                                      ;DSIN

SETF /MR  QIN0  QIN1 /QIN2 /QIN3 /NRZIN         ;VECTOR 11
CLOCKF CLK                                      ;DSIN

SETF /MR  QIN0 /QIN1 /QIN2 /QIN3  NRZIN         ;VECTOR 12, NHDB
CLOCKF CLK                                      ;/DSIN


TRACE_OFF
```

```
Title     HDB3 decoder
Pattern   HDB3DE.pds
Revision A
Author    Cindy Lee
Company   Monolithic Memories Inc., Santa Clara, Ca
Date      3/12/87

CHIP HDB3DE PAL20RS4

;PIN1 PIN2 PIN3 PIN4 PIN5 PIN6
CLK   NC   PHDB NHDB  MR   NC

;PIN7 PIN8 PIN9 PIN10 PIN11 PIN12
NC    NC   NC   NC    NC    GND

;PIN13 PIN14 PIN15  PIN16 PIN17 PIN18
/OE    ERROR VERROR  NC    Q3    Q2

;PIN19 PIN20 PIN21 PIN22 PIN23 PIN24
Q1     Q0    NC    DSOUT RESET VCC


; INPUT SIGNALS
;   CLK    : EXTERNAL CLOCK
;   PHDB   : POSITIVE HDB3 SIGNAL
;   NHDB   : NEGATIVE HDB3 SIGNAL
;   MR     : MASTER RESET
;   /OE    : ASSERTIVE LOW OUTPUT ENABLE

; OUTPUT SIGNALS
;   ERROR  : ERROR FLAG
;   VERROR : BIPOLAR VIOLATION ERRIR
; Q3, Q2, Q1, Q0 : HDB3 DECODING STATE VARIABLE OUTPUTS
;   DSOUT  : DATA SERIAL OUTPUT
;   RESET  : RESET SIGNAL


EQUATIONS

;THE 20RS4 GENERATES ONE SIGNAL, TWO ERROR FLAGS
;AND A HDB3 DECODING STATE MACHINE

;(1) A RESET SIGNAL IS GENERATED TO CLEAR
;    THE 4-BIT SHIFT REGISTER

RESET  =  Q0 * Q2 * /Q3 * /PHDB * NHDB    ; STATE 8 OR STATE 9
                                          ; WITH /PHDB * NHDB
       +  Q1 * /Q2 * Q3 * PHDB * /NHDB    ; STATE 3 OR STATE 4
                                          ; WITH PHDB * /NHDB

;(2) A VERROR SIGNAL IS GENERATED TO INDICATE THE VIOLATION ERROR

VERROR =  /Q0 *  Q1 *  Q2 *  Q3           ; STATE 11
       +   Q0 * /Q1 *  Q2 *  Q3           ; STATE 12
```

```
CLOCKF CLK

SETF /MR  PHDB  NHDB      ;VECTOR 1, STATE 0, 0000
CLOCKF CLK                ;ERROR

SETF /MR /PHDB /NHDB      ;VECTOR 2, STATE 0, 0000
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 3, STATE 0, 0000
CLOCKF CLK

SETF /MR  PHDB /NHDB      ;VECTOR 4, STATE 1, 1000
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 5, STATE 2, 1001
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 6, STATE 3, 1011
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 7, STATE 4, 1010
CLOCKF CLK

SETF /MR  PHDB /NHDB      ;VECTOR 8, STATE 5, 0010
CLOCKF CLK                          ;RESET

SETF /MR /PHDB  NHDB      ;VECTOR 9, STATE 6, 0110
CLOCKF CLK

SETF /MR  PHDB /NHDB      ;VECTOR 10, STATE 1, 1000
CLOCKF CLK

SETF /MR /PHDB  NHDB      ;VECTOR 11, STATE 6, 0110
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 12, STATE 7, 0100
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 13, STATE 8, 0101
CLOCKF CLK

SETF /MR /PHDB  NHDB      ;VECTOR 14, STATE 10, 1111
CLOCKF CLK                          ;RESET

SETF /MR  PHDB /NHDB      ;VECTOR 15, STATE 1, 1000
CLOCKF CLK

SETF /MR /PHDB  NHDB      ;VECTOR 16, STATE 6, 0110
CLOCKF CLK

SETF /MR  PHDB /NHDB      ;VECTOR 17, STATE 1, 1000
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 18, STATE 2, 1001
CLOCKF CLK
```

```
;(3) A ERROR SIGNAL IS GENERATED WHEN PHDB AND NHDB BOTH ARE HIGH
;    AT THE SAME TIME,

ERROR  =  NHDB * PHDB                          ; BOTH PHDB AND NHDB
                                               ; ARE HIGH

;(4) A DATA SERIAL OUTPUT SIGNAL IS GENERATED

DSOUT  =  NHDB
       +  PHDB

;(5) THE HDB3 DECODING STATE MACHINE

Q3     := /NHDB * /PHDB * /Q2 * Q3 * /MR       ; MSB STAE VARIABLE
        + /NHDB * PHDB * /Q0 * /Q3 * /MR
        + /NHDB * /Q0 * Q1 * /Q2 * /Q3 * /MR
        + /NHDB * /Q0 * Q1 * Q2 * Q3 * /MR
        + /NHDB * /Q1 * /Q2 * Q3 * /MR
        + /NHDB * PHDB * Q0 * Q2 * /MR
        + NHDB * /PHDB * Q2 * /Q3 * /MR
        + NHDB * /PHDB * Q0 * Q2 * /MR

Q2     := /NHDB * PHDB * /Q0 * Q1 * /Q2 * /Q3 * /MR
        + /NHDB * PHDB * /Q0 * Q1 * Q2 * Q3 * /MR
        + /NHDB * PHDB * /Q1 * /Q2 * Q3 * /MR
        + NHDB * /PHDB * /Q0 * /Q3 * /MR
        + NHDB * /PHDB * Q1 * Q3 * /MR
        + NHDB * /PHDB * /Q2 * Q3 * /MR
        + /PHDB * Q2 * /Q3 * /MR
        + /PHDB * Q0 * Q2 * /MR

Q1     := /NHDB * /PHDB * Q0 * Q3 * /MR
        + /NHDB * PHDB * /Q2 * Q3 * /MR
        + /NHDB * PHDB * /Q0 * Q1 * /Q2 * /MR
        + /NHDB * PHDB * /Q0 * Q1 * Q3 * /MR
        + NHDB * /PHDB * /Q0 * /Q2 * /MR
        + NHDB * /PHDB * /Q0 * Q1 * Q3 * /MR
        + /PHDB * Q0 * /Q2 * Q3 * /MR
        + /PHDB * Q1 * /Q2 * Q3 * /MR
        + /PHDB * Q0 * Q2 * /Q3 * /MR


Q0     := /NHDB * /PHDB * /Q1 * /Q2 * Q3 * /MR
        + NHDB * /PHDB * Q2 * /Q3 * /MR        ; LSB STATE VARIABLE
        + /PHDB * /Q1 * Q2 * /Q3 * /MR
        + /PHDB * Q0 * Q2 * /Q3 * /MR
        + NHDB * /PHDB * Q0 * Q2 * /MR

SIMULATION

TRACE_ON MR CLK PHDB NHDB DSOUT Q3 Q2 Q1 Q0 RESET VERROR ERROR

SETF  OE MR  PHDB  NHDB      ;RESET CONDITION
```

```
SETF /MR /PHDB /NHDB      ;VECTOR 19, STATE 3, 1011
CLOCKF CLK

SETF /MR  PHDB /NHDB      ;VECTOR 20, STATE 5, 0010
CLOCKF CLK                          ;RESET

SETF /MR /PHDB  NHDB      ;VECTOR 21, STATE 6, 0110
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 22, STATE 7, 0100
CLOCKF CLK

SETF /MR /PHDB /NHDB      ;VECTOR 23, STATE 8, 0101
CLOCKF CLK

SETF /MR /PHDB  NHDB      ;VECTOR 24, STATE 9, 0111
CLOCKF CLK

SETF /MR /PHDB  NHDB      ;VECTOR 25, STATE 10, 1111
CLOCKF CLK                          ;RESET

TRACE_OFF
```

ZPAL™ Devices Implement D4 Frame Synchronization

AN-170

With the rapid increase in popularity of T1 trunk links following the deregulation of AT&T, the transparent nature of the 24 data channels must be supported by handling of the framing bits. These must comply with certain standards, the most common of which is known as D4.

T1-Carriers

More than 100 million miles of T-carrier lines are installed in the U.S. and Canada. They are full-duplex digital communication links with a specific bit rate and format. T-carrier lines' transmission quality and reliability exceeds that of analog lines. T-carrier lines are available in a range of transmission rates between 1.544 Mbps and 274 Mbps, designated T1, T1C, T2, T3, and T4.

The most common standard, T1 can be considered a building block for the others. T1 transmits at 1.544 Mbps. AT&T specifications define D1, D1D, D2, D3, D4 and ESF framing structures for T1 carriers. These standards specify the framing-bit pattern and frame synchronization. D1 was established in 1969 and has been superseded by the later standards. D1D, D2, D3, and D4 framing formats group twelve frames into a superframe. Today, the D4 framing format is the most prevalent. The future format is ESF, also called Fe, which groups twenty-four frames into an extended superframe. It is presently under definition, and will be supported in the future.

D4 Framing Format

The T1 standard for D4 framing defines a superframe as a group of twelve frames. A frame consists of twenty-four 8-bit channels plus one framing bit for a total of 193 bits. The 8-bit channel consists of one polarity bit and seven magnitude bits; however, in frames six and twelve, the eighth bit in each channel carries signaling information. Since this bit cannot transmit user data (either voice or data) it is called robbed-bit signaling. The signaling bit transmits requests for services and calling information. At the 1.544 Mbps transmission rate, 8 kbps are overhead for framing; so the user transmission rate is 1.536 Mbps. Because the user transmission rate is divided into twenty-four channels, each channel transmits at 64 kbps.

A frame bit should occur every 193rd bit in the data stream. Frame detection occurs when a sequence of sampled frame bits matches the frame pattern. However, occasionally data can produce the same pattern. False framing occurs when the frame synchronization circuitry mistakes data bits for the framing bits.

A frame-bit error occurs when the incoming frame bit does not match the expected framing bit. If two out of four consecutive frame bits are in error, then frame synchronization is "lost." Once frame synchronization is "lost," a search for the frame pattern is started. AT&T T1 standards specify the maximum time to search and re-synchronize as 50 ms. This time period is called the reframe time.

For the D4 frame format, the framing bits are divided into two groups: Ft and Fs. The terminal framing bit (Ft) identifies frame boundaries. The signaling frame bit (Fs) identifies the superframe boundary and the frames containing signaling bits. Ft and Fs are interleaved to create the 12-bit framing pattern shown in Table 1.This pattern is repeated every twelve frames.

FRAME	TERMINAL FRAMING Ft	SIGNALING FRAMING Fs	SUPER-FRAMING SF
1	1	—	1
2	—	0	0
3	0	—	0
4	—	0	0
5	1	—	1
6	—	1	1
7	0	—	0
8	—	1	1
9	1	—	1
10	—	1	1
11	0	—	0
12	—	0	0

Table 1. Framing Pattern

Hardware Implementation

The frame synchronization logic was implemented in PAL devices because current single-chip solutions do not offer enough flexibility in line coding. Using PAL devices, this design has the flexibility to interface various line coding devices.

The PAL devices implement the frame synchronization logic, which can be divided into three sections. The first section detects the D4 framing pattern. The second section searches for frame synchronization. The third section provides the superframe timing. These are implemented using four CMOS PAL devices, a shift register, and a counter as shown in Figure 1.

The frame detection hardware samples every 193rd bit of the data stream. The data is sampled on the positive edge of T1_CK. Either a shift register or a state machine can be used to sample the framing bits. The data stream is shifted into a (193x23)-bit shift register with taps every 193 bits. The taps are fed into two identical Frame Pattern Detection (FPD) PAL devices. The more taps sampled, the less likely a data pattern will reproduce the framing pattern. In this design, twenty-four taps ensure reasonable prevention of false framing.

Each FPD device monitors twelve framing bits. If the framing pattern is found, a FP_DETB signal is generated. From these signals, the controller determines the correct framing pattern. The FPD's FRM1B indicates the start of a superframe.

The first FPD's inputs are the first eleven taps from the shift register plus the data input stream. The second FDP monitors the remaining taps from the shift register as shown in Figure 1.

The control section is a controller and a counter, which search for frame synchronization. The controller, SYNC PAL, uses the 193-bit counter to determine if the frame detection signals are spaced every 193 bits. First the controller searches for the first framing pattern. During the search operation, SYNC PAL waits for the FPD's frame detection signals. When a framing pattern is found, the counter is reset. If 193 bits later, the next expected framing bit is found, the controller is "in sync." It continues checking the framing pattern every 193 bits. If the framing pattern is not found, it assumes that the incoming framing bit is in error. If the next four framing bits are correct, the controller remains "in sync." If, during this time, a second frame-bit error occurs, the framing is "lost." The controller, then, asserts the lost frame signal, NO_FRM. Once the framing is "lost," the controller begins searching for the correct frame pattern. While the controller searches for the correct frame pattern, the previous frame synchronization is maintained until the controller re-synchronizes.

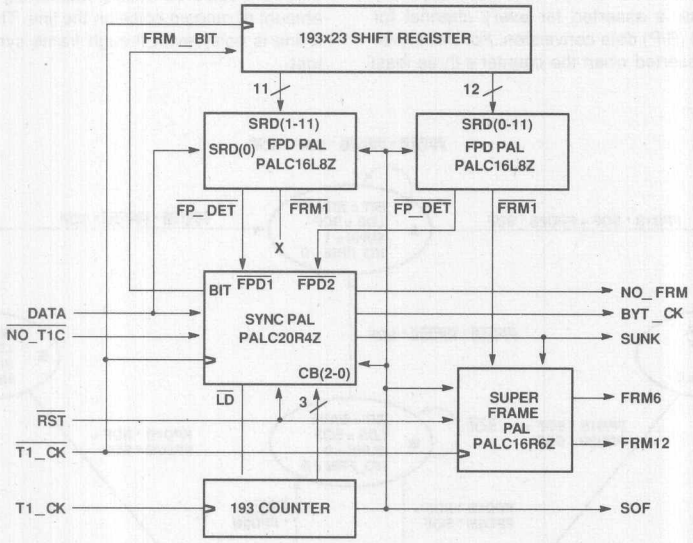


Figure 1. Frame Synchronization Logic

Figure 2 shows the sync. state machine for the controller. Signals FPD1B and FPD2B (from the FPDs) indicate a T1 frame pattern detection, while SOF (from the counter) indicates 193 counts. Beginning in states F and G, the controller searches for the frame pattern. Once the framing has been found, the state machine enters state A. At state A SUNK is asserted, which indicates "in sync." The state machine remains in state A until a single frame-bit error occurs. Upon the first frame-bit error, the state machine enters state B. If the next four framing bits are correct, the state machine sequences through states C, D, and E, which returns to state A. However, if during this time, a second frame bit error occurs, the state machine enters state F. Loss of T1 carrier (NO_T1CB) or reset (RSTB) forces the state machine into the searching state.

SYNC PAL performs two other miscellaneous functions. When a single framing-bit error is encountered, the device corrects the bit. The corrected version of the input data stream, BIT, is fed into the shift register. The device also generates the BYT_CK. A single pulse is asserted for every channel for external serial-to-parallel (S/P) data conversion. For this implementation BYT_CK is asserted when the counter's three least significant bits are all zeros. Depending on the requirements of the S/P converter, any combination of the counter's three least significant bits could generate BYT_CK for the correct alignment with the channel sample.

SUPER FRAME PAL generates the superframe timing. It generates FRM_6 and FRM_12 by counting the frames. These signals indicate the frames containing the signaling bits. The CMOS PAL device implements a 4-bit counter. When the frame is "in sync," SOF increments the counter, while FRM_1 clears it. If the controller is "out of frame sync," the counter continues counting, based on the last known frame position.

Design Recommendations

While this design meets the D4 specification, one design improvement might be useful. A signal flagging single-framing bit errors could be added. Monitoring this signal indicates the amount of random noise on the line. This signal could warn that a line is noisy even though frame synchronization was never lost.

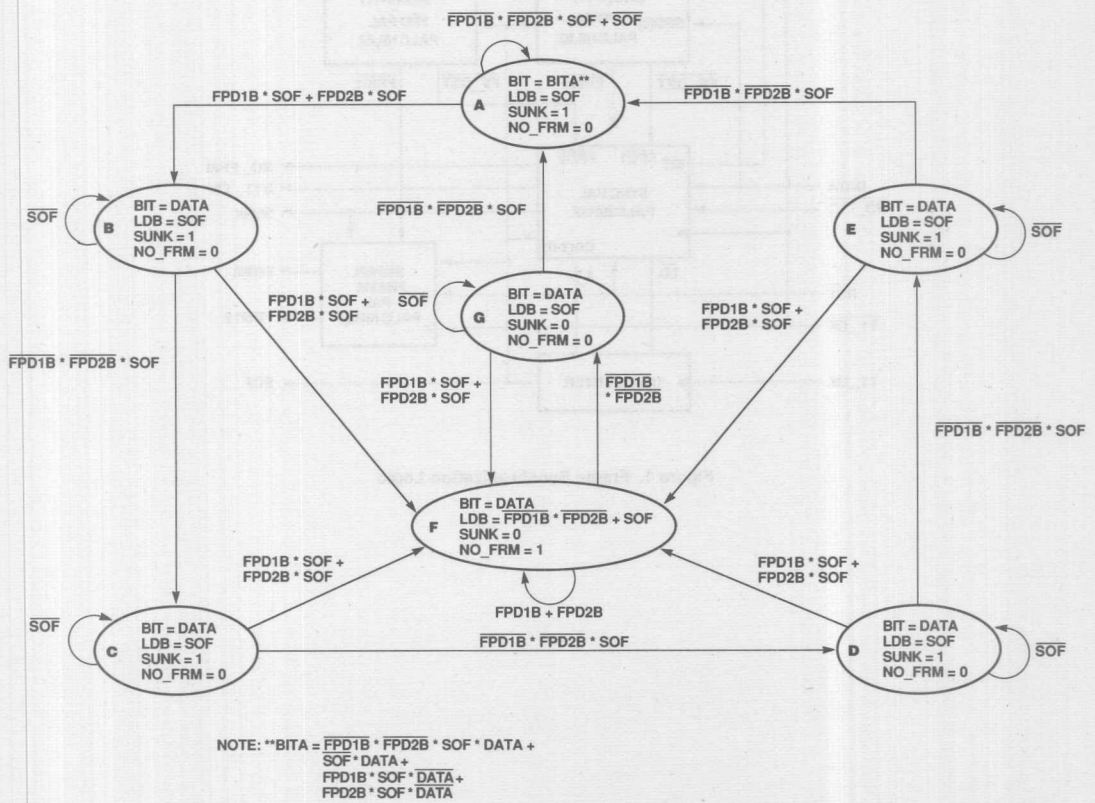


Figure 2. Sync State Machine

```
TITLE FRAME_DETECTION_PAL
PATTERN  T1 FRAME DETECTION PAL FOR T1 INTERFACE
REVISION P1.04
AUTHOR STEVE PATTERSON AND THERESA SHAFER
COMPANY
DATE 6/4/87

;  This PAL device monitors 12 193rd bits in the incoming T1 NRZ data s
;  It detects any valid Frame Pattern (start of any Frame) and the start
;  of Frame 1.

CHIP FPD PAL16L8

;PINS
;1       2        3        4        5        6        7        8        9        10
SRD0     SRD1     SRD2     SRD3     SRD4     SRD5     SRD6     SRD7     SRD8     GND
;11      12       13       14       15       16       17       18       19       20
SRD9     FRM1B    SRD10    SRD11    NC       FRM1_6B  NC       SOF      FP_DETB  VCC

; INPUTS:        SRD(11-0)        FRAMING BITS
;                 SOF             LAST KNOWN START OF FRAME

; OUTPUTS:       FP_DETB          ACTIVE LOW SIGNAL INDICATING FRAMING
;                                  PATTERN DETECTED
;                 FRM1B           ACTIVE LOW SIGNAL INDICATING START OF
;                                  FRAME 1
;                 FRM1_6B         ACTIVE LOW SIGNAL INDICATING FRAME 1
;                                  TO 6 DETECTED

;FRAMING PATTERNS
;                        S S S S S S S S S S S S
;                        R R R R R R R R R R R R
;                        D D D D D D D D D D D D
;                        1 1 9 8 7 6 5 4 3 2 1 0
;                        1 0
;       FRM_1    =  [ 1,0,0,0,1,1,0,1,1,1,0,0 ];
;       FRM_2    =  [ 0,1,0,0,0,1,1,0,1,1,1,0 ];
;       FRM_3    =  [ 0,0,1,0,0,0,1,1,0,1,1,1 ];
;       FRM_4    =  [ 1,0,0,1,0,0,0,1,1,0,1,1 ];
;       FRM_5    =  [ 1,1,0,0,1,0,0,0,1,1,0,1 ];
;       FRM_6    =  [ 1,1,1,0,0,1,0,0,0,1,1,0 ];
;       FRM_7    =  [ 0,1,1,1,0,0,1,0,0,0,1,1 ];
;       FRM_8    =  [ 1,0,1,1,1,0,0,1,0,0,0,1 ];
;       FRM_9    =  [ 1,1,0,1,1,1,0,0,1,0,0,0 ];
;       FRM_10   =  [ 0,1,1,0,1,1,1,0,0,1,0,0 ];
;       FRM_11   =  [ 0,0,1,1,0,1,1,1,0,0,1,0 ];
;       FRM_12   =  [ 0,0,0,1,1,0,1,1,1,0,0,1 ];

EQUATIONS

/FRM1B = SOF * SRD11 * /SRD10 * /SRD9 * /SRD8 * SRD7 * SRD6 * /SRD5
              * SRD4 * SRD3 * SRD2 * /SRD1 * /SRD0

/FRM1_6B = SRD11 * /SRD10 * /SRD9 * /SRD8 * SRD7 * SRD6 * /SRD5  ; FRM 1
              * SRD4 * SRD3 * SRD2 * /SRD1 * /SRD0
```

```
        + /SRD11 * SRD10 * /SRD9 * /SRD8 * /SRD7 * SRD6 * SRD5    ; FRM 2
                 * /SRD4 * SRD3 * SRD2 * SRD1 * /SRD0
        + /SRD11 * /SRD10 * SRD9 * /SRD8 * /SRD7 * /SRD6 * SRD5   ; FRM 3
                 * SRD4 * /SRD3 * SRD2 * SRD1 * SRD0
        + SRD11 * /SRD10 * /SRD9 * SRD8 * /SRD7 * /SRD6 * /SRD5   ; FRM 4
                 * SRD4 * SRD3 * /SRD2 * SRD1 * SRD0
        + SRD11 * SRD10 * /SRD9 * /SRD8 * SRD7 * /SRD6 * /SRD5    ; FRM 5
                 * /SRD4 * SRD3 * SRD2 * /SRD1 * SRD0
        + SRD11 * SRD10 * SRD9 * /SRD8 * /SRD7 * SRD6 * /SRD5     ; FRM 6
                 * /SRD4 * /SRD3 * SRD2 * SRD1 * /SRD0

/FP_DETB = /FRM1_6B                                               ; FRM 1-6
        + /SRD11 * SRD10 * SRD9 * SRD8 * /SRD7 * /SRD6 * SRD5     ; FRM 7
                 * /SRD4 * /SRD3 * /SRD2 * SRD1 * SRD0
        + SRD11 * /SRD10 * SRD9 * SRD8 * SRD7 * /SRD6 * /SRD5     ; FRM 8
                 * SRD4 * /SRD3 * /SRD2 * /SRD1 * SRD0
        + SRD11 * SRD10 * /SRD9 * SRD8 * SRD7 * SRD6 * /SRD5      ; FRM 9
                 * /SRD4 * SRD3 * /SRD2 * /SRD1 * /SRD0
        + /SRD11 * SRD10 * SRD9 * /SRD8 * SRD7 * SRD6 * SRD5      ; FRM 10
                 * /SRD4 * /SRD3 * SRD2 * /SRD1 * /SRD0
        + /SRD11 * /SRD10 * SRD9 * SRD8 * /SRD7 * SRD6 * SRD5     ; FRM 11
                 * SRD4 * /SRD3 * /SRD2 * SRD1 * /SRD0
        + /SRD11 * /SRD10 * /SRD9 * SRD8 * SRD7 * /SRD6 * SRD5    ; FRM 12
                 * SRD4 * SRD3 * /SRD2 * /SRD1 * SRD0

; . . . . . .
; . . . . . .

SIMULATION

TRACE_ON
SOF SRD11 SRD10 SRD9 SRD8 SRD7 SRD6 SRD5 SRD4 SRD3 SRD2 SRD1 SRD0
FP_DETB FRM1B FRM1_6B

SETF SOF

SETF                                                              ; FRAME 1
 SRD11 /SRD10 /SRD9 /SRD8  SRD7  SRD6 /SRD5  SRD4  SRD3  SRD2 /SRD1 /SRD0
SETF                                                              ; FRAME 2
/SRD11 SRD10 /SRD9 /SRD8 /SRD7  SRD6  SRD5 /SRD4  SRD3  SRD2  SRD1 /SRD0
SETF                                                              ; FRAME 3
/SRD11 /SRD10  SRD9 /SRD8 /SRD7 /SRD6  SRD5  SRD4 /SRD3  SRD2  SRD1  SRD0
SETF                                                              ; FRAME 4
 SRD11 /SRD10 /SRD9  SRD8 /SRD7 /SRD6 /SRD5  SRD4  SRD3 /SRD2  SRD1  SRD0
SETF                                                              ; FRAME 5
 SRD11  SRD10 /SRD9 /SRD8  SRD7 /SRD6 /SRD5 /SRD4  SRD3  SRD2 /SRD1  SRD0
SETF                                                              ; FRAME 6
 SRD11  SRD10  SRD9 /SRD8 /SRD7  SRD6 /SRD5 /SRD4 /SRD3  SRD2  SRD1 /SRD0
SETF                                                              ; FRAME 7
/SRD11  SRD10  SRD9  SRD8 /SRD7 /SRD6  SRD5 /SRD4 /SRD3 /SRD2  SRD1  SRD0
SETF                                                              ; FRAME 8
 SRD11 /SRD10  SRD9  SRD8  SRD7 /SRD6 /SRD5  SRD4 /SRD3 /SRD2 /SRD1  SRD0
SETF                                                              ; FRAME 9
 SRD11  SRD10 /SRD9  SRD8  SRD7  SRD6 /SRD5 /SRD4  SRD3 /SRD2 /SRD1 /SRD0
SETF                                                              ; FRAME 10
/SRD11   SRD10   SRD9 /SRD8   SRD7   SRD6   SRD5 /SRD4 /SRD3   SRD2 /SRD1 /SRD0
```

```
SETF                                                          ; FRAME 11
/SRD11 /SRD10  SRD9  SRD8 /SRD7  SRD6  SRD5  SRD4 /SRD3 /SRD2  SRD1 /SRD0
SETF                                                          ; FRAME 12
/SRD11 /SRD10 /SRD9  SRD8  SRD7 /SRD6  SRD5  SRD4  SRD3 /SRD2 /SRD1  SRD0


SETF                                                          ; NOT FRAMING PATTERN
/SRD11 /SRD10 /SRD9 /SRD8 /SRD7 /SRD6 /SRD5 /SRD4 /SRD3 /SRD2 /SRD1 /SRD0
SETF                                                          ; NOT FRAMING PATTERN
/SRD11 /SRD10  SRD9 /SRD8 /SRD7 /SRD6 /SRD5 /SRD4 /SRD3 /SRD2 /SRD1 /SRD0
SETF                                                          ; NOT FRAMING PATTERN
/SRD11 /SRD10 /SRD9 /SRD8 /SRD7 /SRD6 /SRD5 /SRD4 /SRD3 /SRD2 /SRD1 /SRD0
SETF                                                          ; NOT FRAMING PATTERN
/SRD11  SRD10  SRD9 /SRD8  SRD7 /SRD6 /SRD5 /SRD4 /SRD3 /SRD2 /SRD1 /SRD0
SETF                                                          ; NOT FRAMING PATTERN
 SRD11  SRD10  SRD9  SRD8 /SRD7  SRD6  SRD5  SRD4  SRD3  SRD2  SRD1  SRD0
SETF                                                          ; NOT FRAMING PATTERN
 SRD11  SRD10  SRD9  SRD8  SRD7  SRD6  SRD5  SRD4  SRD3  SRD2  SRD1  SRD0

TRACE_OFF
```

```
PALASM SIMULATION, V2.22 - MARKET RELEASE (11-19-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title    : FRAME_DETECTION_PAL     Author   : STEVE PATTERSON AND THER
Pattern  : T1 FRAME DETECTION PAL FCompany  :
Revision : P1.04                   Date     : 6/4/87


FPD
Page :  1
          gggggggggg gggggggggg
 SOF      HHHHHHHHHH HHHHHHHHH
 SRD11    XHLLHHHLHH LLLLLLLHH
 SRD10    XLHLLHHHLH HLLLLLHHH
 SRD9     XLLHLLHHHL HHLLHLHHH
 SRD8     XLLLHLLHHH LHHLLLLHH
 SRD7     XHLLLHLLHH HLHLLLHLH
 SRD6     XHHLLLHLLH HHLLLLLHH
 SRD5     XLHHLLLHLL HHHLLLLHH
 SRD4     XHLHHLLLHL LHHLLLLHH
 SRD3     XHHLHHLLLH LLHLLLLHH
 SRD2     XHHHLHHLLL HLLLLLLHH
 SRD1     XLHHHLHHLL LHLLLLLHH
 SRD0     XLLHHHLHHL LLHLLLLHH
 FP_DETB  XLLLLLLLLL LLLHHHHHH
 FRM1B    XLHHHHHHHH HHHHHHHHH
 FRM1_6B  XLLLLLLHHH HHHHHHHHH
```

```
TITLE SYNC_PAL
PATTERN T1 FRAME SYNC PAL FOR T1 INTERFACE
REVISION P1.06
AUTHOR STEVE PATTERSON AND THERESA SHAFER
COMPANY
DATE 6/4/87

;  This PAL device decides whether the T1 Interface is in Frame Sync,
;  provisional Sync, or Out of Sync.  It controls the Frame Sync
;  process.

CHIP SYNC PAL20R4

;PINS
;1       2       3       4       5       6       7       8
T1_CKB   RSTB    NO_T1CB DATA    FPD1B   FPD2B   BC2     BC1
;9       10      11      12
BC0      SOF     NC      GND

;13      14      15      16      17      18      19      20
OEB      NC      NO_FRM  SUNK    LDB     Q0      Q1      Q2
;21      22      23      24
BYT_CK   BIT     NC      VCC

;INPUTS:T1_CKB           ACTIVE LOW EXTERNAL T1 CLOCK
;       RSTB             ACTIVE LOW MASTER RESET
;       NO_T1CB          NO T1 SIGNAL ACTIVE LOW INPUT
;       FPD(1-2)B        ACTIVE LOW FRAME PATTERN DETECT INPUTS
;       BC(2-0)          193 COUNTER 3 LSBs INPUTS USED FOR BYT_CK
;       SOF              LAST KNOWN START OF FRAME
;       DATA             INPUT DATA STREAM
;       OEB              ACTIVE LOW OUTPUT ENABLE INPUT

;OUTPUTS:BIT             NEXT T1 BIT STREAM
;       LDB              ACTIVE LOW LOAD COUNTER SIGNAL
;       SUNK             IN SYNC STATE AND NO FRAME BIT ERROR
;       NO_FRM           FRAME LOST, IN SEARCH STATE
;       Q(2-0)           STATE VARIABLES
;       BYT_CK           8-BIT CHANNEL SAMPLE CLOCK

EQUATIONS

;STATE MACHINE OUTPUTS AND STATE VARIABLES

/BIT    = DATA * /Q2 * /Q1 * Q0 * FPD1B * SOF
        + DATA * /Q2 * /Q1 * Q0 * FPD2B * SOF
        + /DATA * /SOF
        + /DATA * /FPD1B * /FPD2B
        + /DATA * Q2
        + /DATA * Q1
        + /DATA * /Q0

/LDB    := Q2 * Q1 * Q0 * /FPD1B * /FPD2B + SOF

/SUNK   = /Q2 * /Q1 * /Q0 + Q1 * Q0             ; IN FRAME SYNC
```

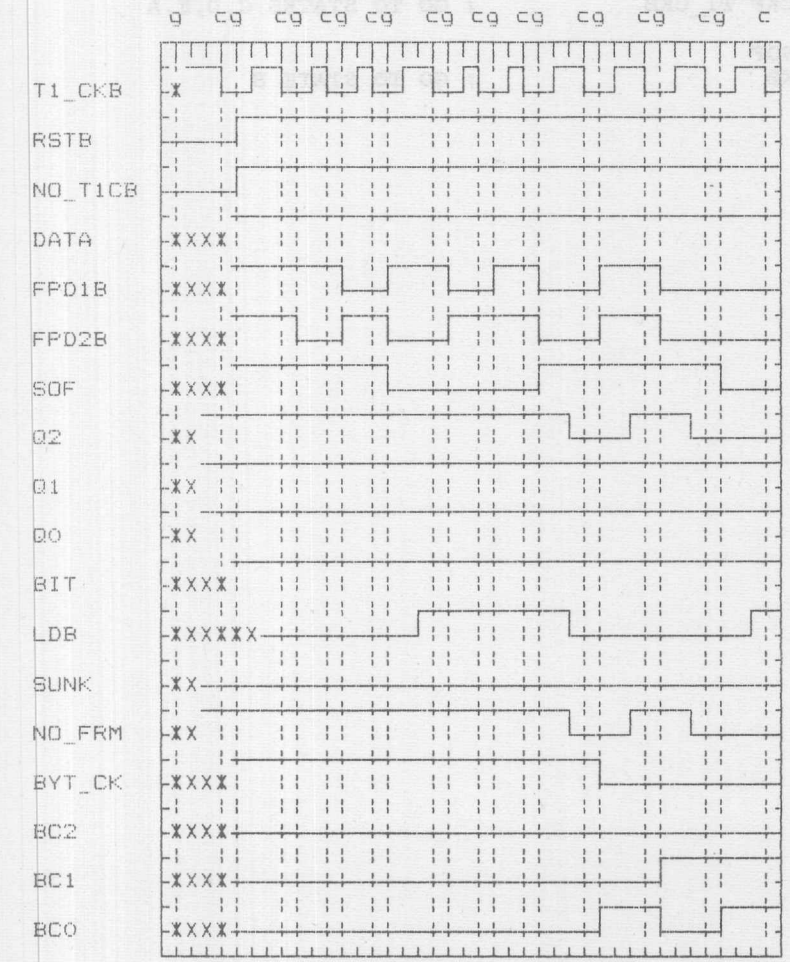
```
     /NO_FRM = /Q2 + /Q1 + /Q0                ; SEARCHING FOR FRAME SYNC

; USING THE FOLLOWING STATE ASSIGNMENTS
;       A = 001          IN SYNC STATE
;       B = 010          SINGLE FRAME BIT ERROR STATE
;       C = 110
;       D = 100
;       E = 101
;       F = 111          SEARCHING STATE
;       G = 011          SEARCHING STATE
;       H = 000          UNUSED STATE - GOES TO F

/Q2     := /FPD2B * /FPD1B * Q1 * Q0 * RSTB * NO_T1CB
        + /Q2 * Q1 * /SOF * RSTB * NO_T1CB
        + /Q2 * /Q1 * Q0 * RSTB * NO_T1CB
        + /FPD2B * /FPD1B * Q0 * SOF * RSTB * NO_T1CB

/Q1     := /Q1 * Q0 * /SOF * RSTB * NO_T1CB
        + Q2 * /Q1 * /SOF * RSTB * NO_T1CB
        + /FPD2B * /FPD1B * Q2 * /Q0 * SOF * RSTB * NO_T1CB
        + /FPD2B * /FPD1B * Q2 * /Q1 * RSTB * NO_T1CB
        + /FPD2B * /FPD1B * /Q2 * Q0 * SOF * RSTB * NO_T1CB

/Q0     := Q1 * /Q0 * /SOF * RSTB * NO_T1CB
        + Q2 * /Q0 * /SOF * RSTB * NO_T1CB
        + FPD1B * /Q2 * /Q1 * Q0 * SOF  * RSTB * NO_T1CB
        + FPD2B * /Q2 * /Q1 * Q0 * SOF  * RSTB * NO_T1CB
        + /FPD2B * /FPD1B * Q1 * /Q0 * RSTB * NO_T1CB

; BYTE CLOCK

/BYT_CK = BC2 + BC1 + BC0

; ..............................................................
; ..............................................................

SIMULATION

TRACE_ON T1_CKB RSTB NO_T1CB DATA FPD1B FPD2B SOF
Q2 Q1 Q0 BIT LDB SUNK NO_FRM
BYT_CK BC2 BC1 BC0

SETF /OEB                          ; ENABLE OUTPUT
     /RSTB /NO_T1CB                ; RESET REGISTERS
CLOCKF T1_CKB

SETF RSTB NO_T1CB                  ; RESET REGISTERS
     DATA
     /BC2 /BC1 /BC0                ; STATE MACHINE - GO TO STATE F
     FPD1B FPD2B SOF
CLOCKF T1_CKB

SETF FPD1B /FPD2B SOF
CLOCKF T1_CKB                      ; LOOPS AT STATE F
```

```
SETF /FPD1B FPD2B SOF
CLOCKF T1_CKB                    ; LOOPS AT STATE F

SETF FPD1B /FPD2B /SOF
CLOCKF T1_CKB                    ; LOOPS AT STATE F

SETF /FPD1B FPD2B /SOF
CLOCKF T1_CKB                    ; LOOPS AT STATE F

SETF FPD1B FPD2B /SOF
CLOCKF T1_CKB                    ; LOOPS AT STATE F

SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE G

SETF FPD1B FPD2B SOF
     /BC2 /BC1 BC0
CLOCKF T1_CKB                    ; GO TO STATE F

SETF /FPD1B /FPD2B
     /BC2 BC1 /BC0
CLOCKF T1_CKB                    ; GO TO STATE G

SETF /SOF
     /BC2 BC1 BC0
CLOCKF T1_CKB                    ; LOOP AT STATE G
SETF FPD1B /FPD2B SOF
     BC2 /BC1 /BC0
CLOCKF T1_CKB                    ; GO TO STATE F

SETF /FPD1B /FPD2B /SOF
     BC2 /BC1 BC0
CLOCKF T1_CKB                    ; GO TO STATE G
SETF /FPD1B /FPD2B SOF
     BC2 BC1 /BC0
CLOCKF T1_CKB                    ; GO TO STATE A

SETF /SOF
     BC2 BC1 BC0
CLOCKF T1_CKB                    ; LOOP AT STATE A
SETF /FPD1B /FPD2B SOF
     /BC2 /BC1 /BC0
CLOCKF T1_CKB                    ; LOOP AT STATE A

SETF /FPD1B FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE B
SETF FPD1B FPD2B /SOF
     /BC2 /BC1 BC0
CLOCKF T1_CKB                    ; LOOP AT STATE B
SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE C

SETF /SOF
CLOCKF T1_CKB                    ; LOOP AT STATE C
```

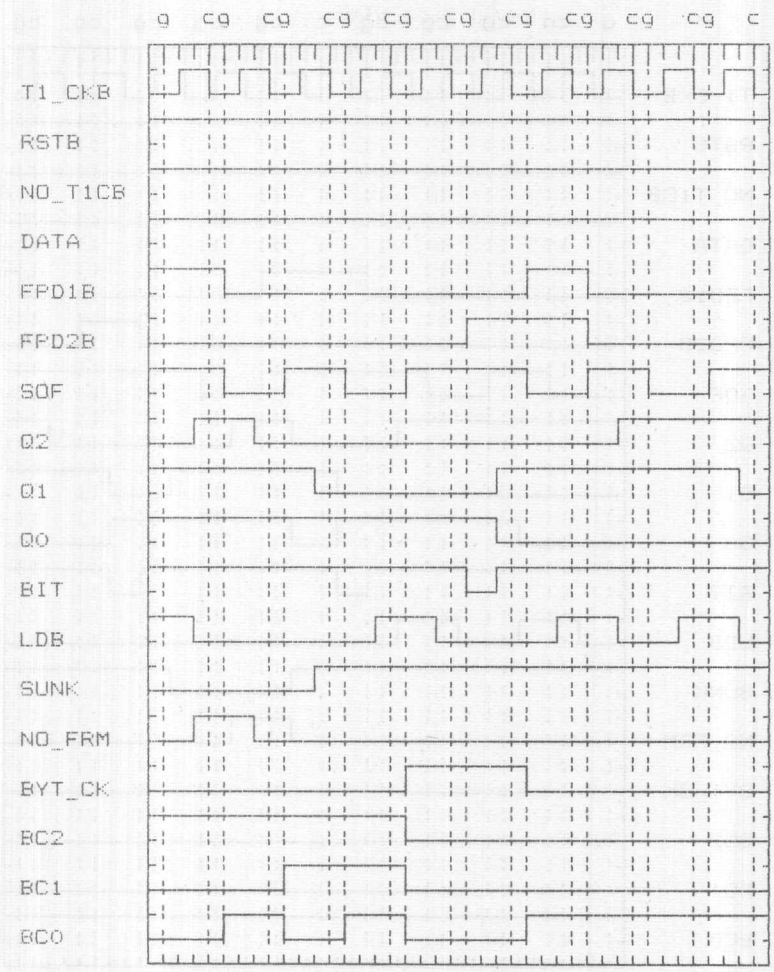
```
CLOCKF T1_CKB                    ; GO TO STATE D

SETF /SOF
CLOCKF T1_CKB                    ; LOOP AT STATE D
SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE E

SETF /SOF
CLOCKF T1_CKB                    ; LOOP AT STATE E
SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE A

SETF FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE B
SETF FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE F

SETF /FPD1B /FPD2B /SOF
CLOCKF T1_CKB                    ; GO TO STATE G
SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE A
SETF /FPD1B FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE B
SETF /FPD1B /FPD2B SOF
FOR I:=1 TO 1 DO BEGIN
        CLOCKF T1_CKB            ; GO TO STATE C
END
SETF FPD1B SOF
CLOCKF T1_CKB                    ; GO TO STATE F

SETF /FPD1B /FPD2B /SOF
CLOCKF T1_CKB                    ; GO TO STATE G
SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE A
SETF /FPD1B FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE B
SETF /FPD1B /FPD2B SOF
FOR I:=1 TO 2 DO BEGIN
        CLOCKF T1_CKB            ; GO TO STATES C,D
END
SETF FPD1B /SOF
CLOCKF T1_CKB                    ; LOOP AT STATE D
SETF FPD1B SOF
CLOCKF T1_CKB                    ; GO TO STATE F

SETF /FPD1B /FPD2B /SOF
CLOCKF T1_CKB                    ; GO TO STATE G
SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE A
SETF /FPD1B FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE B
SETF /FPD1B /FPD2B SOF
FOR I:=1 TO 3 DO BEGIN
        CLOCKF T1_CKB            ; GO TO STATES C,D,E
```

```
END
SETF FPD1B SOF
CLOCKF T1_CKB                    ; GO TO STATE F

SETF /DATA                       ; INVERT DATA
SETF /FPD1B /FPD2B /SOF
CLOCKF T1_CKB                    ; GO TO STATE G
SETF /FPD1B /FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE A
SETF /FPD1B FPD2B SOF
CLOCKF T1_CKB                    ; GO TO STATE B
SETF /FPD1B /FPD2B SOF
FOR I:=1 TO 4 DO BEGIN
       CLOCKF T1_CKB             ; GO TO STATES C,D,E,A
END
SETF FPD1B SOF
CLOCKF T1_CKB                    ; GO TO STATE B
TRACE_OFF
```

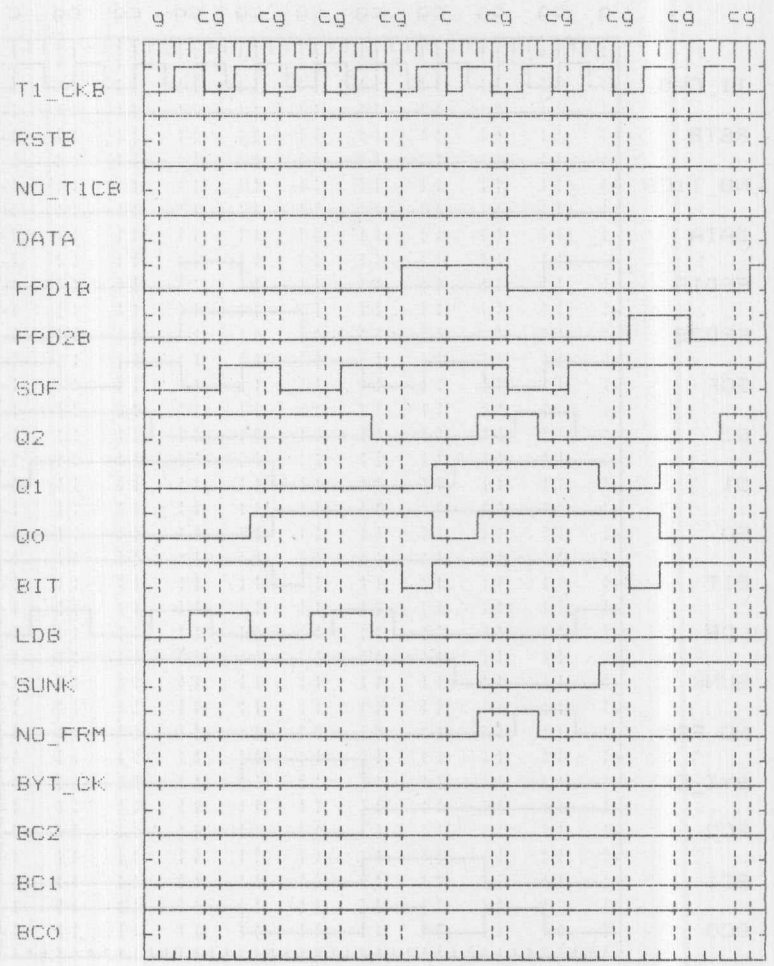
```
PALASM SIMULATION, V2.22 - MARKET RELEASE (11-19-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title     : SYNC_PAL                    Author    : STEVE PATTERSON AND THER
Pattern   : T1 FRAME SYNC PAL FOR T1Company :
Revision  : P1.06                       Date      : 6/4/87

SYNC
Page :  1
```



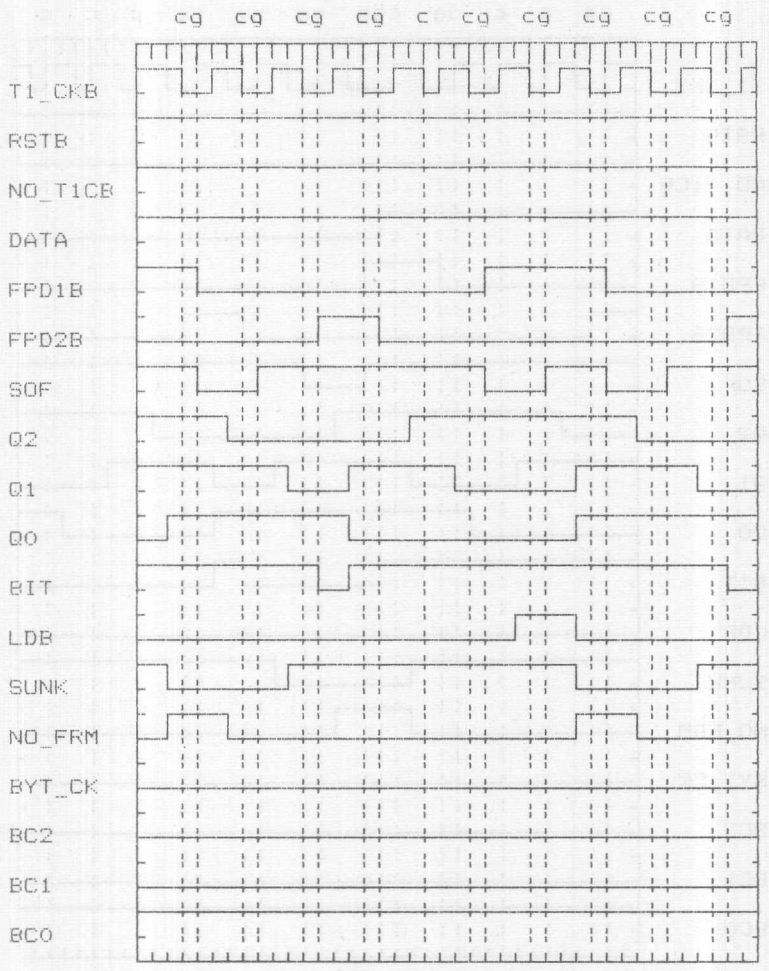
SYNC
Page : 2


T1_CKB
RSTB
NO_T1CB
DATA
FPD1B
FPD2B
SOF
Q2
Q1
Q0
BIT
LDB
SUNK
NO_FRM
BYT_CK
BC2
BC1
BC0

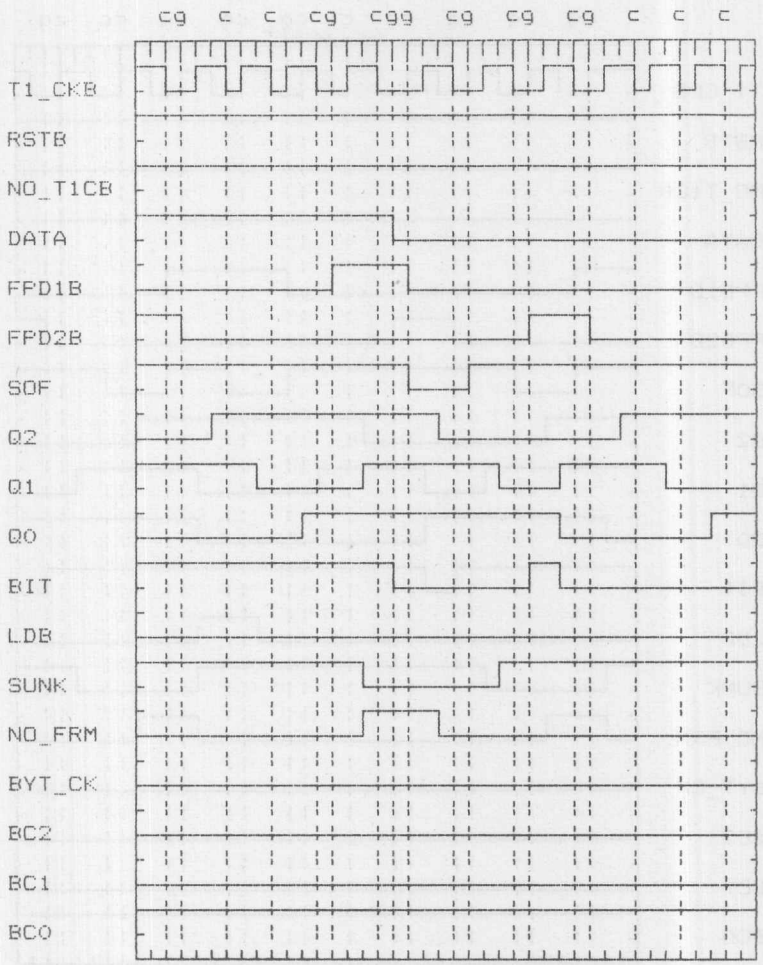
SYNC
Page : 3


g cg cg cg cg c cg cg cg cg cg
T1_CKB
RSTB
NO_T1CB
DATA
FPD1B
FPD2B
SOF
Q2
Q1
Q0
BIT
LDB
SUNK
NO_FRM
BYT_CK
BC2
BC1
BC0

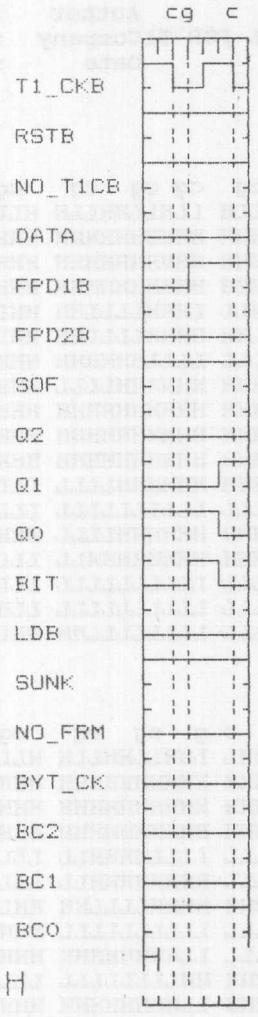
SYNC
Page : 4


cg cg cg cg c cg cg cg cg cg
T1_CKB
RSTB
NO_T1CB
DATA
FPD1B
FPD2B
SOF
Q2
Q1
Q0
BIT
LDB
SUNK
NO_FRM
BYT_CK
BC2
BC1
BC0

SYNC
Page : 5


cg c c cg cgg cg cg cg c c c
T1_CKB
RSTB
NO_T1CB
DATA
FPD1B
FPD2B
SOF
Q2
Q1
Q0
BIT
LDB
SUNK
NO_FRM
BYT_CK
BC2
BC1
BC0

SYNC
Page : 6


cg
c
T1_CKB
RSTB
NO_T1CB
DATA
FPD1B
FPD2B
SOF
Q2
Q1
Q0
BIT
LDB
SUNK
NO_FRM
BYT_CK
BC2
BC1
BC0
H
H

```
PALASM SIMULATION, V2.22 - MARKET RELEASE (11-19-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title    : SYNC_PAL                    Author   : STEVE PATTERSON AND THER
Pattern  : T1 FRAME SYNC PAL FOR T1Company  :
Revision : P1.06                       Date     : 6/4/87


SYNC
Page :  1
          g  cg  cg  cg cg  cg  cg cg  cg   cg  cg  c
 T1_CKB   XHHLLHHLLH LLHLLHHLLH LLHLLHHLLH HLLHHLLHHL
 RSTB     LLLLHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 NO_T1CB  LLLLHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 DATA     XXXXHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 FPD1B    XXXXHHHHHH HLLLHHHHLL LHHHLLLLHH HHLLLLLLLL
 FPD2B    XXXXHHHHLL LHHHLLLLHH HHHHLLLLHH HHLLLLLLLL
 SOF      XXXXHHHHHH HHHHLLLLLL LLLLHHHHHH HHHHHHLLLL
 Q2       XXHHHHHHHH HHHHHHHHHH HHHHHHLLLL HHHHLLLLLL
 Q1       XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 Q0       XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 BIT      XXXXHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 LDB      XXXXXXLLLL LLLLLLHHHH HHHHHHLLLL LLLLLLLLHH
 SUNK     XXLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 NO_FRM   XXHHHHHHHH HHHHHHHHHH HHHHHHLLLL HHHHLLLLLL
 BYT_CK   XXXXHHHHHH HHHHHHHHHH HHHHHHHHLL LLLLLLLLLL
 BC2      XXXXLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC1      XXXXLLLLLL LLLLLLLLLL LLLLLLLLLL LLHHHHHHHH
 BC0      XXXXLLLLLL LLLLLLLLLL LLLLLLLLHH HHLLLLHHHH

SYNC
Page :  2
          g  cg  cg   cg  cg  c g  cg  cg   cg  cg  c
 T1_CKB   LHHLLHHLLH HLLHHLLHHL LHHLLHHLLH HLLHHLLHHL
 RSTB     HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 NO_T1CB  HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 DATA     HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 FPD1B    HHHHLLLLLL LLLLLLLLLL LLLLHHHHLL LLLLLLLLLL
 FPD2B    LLLLLLLLLL LLLLLLLLLL HHHHHHHHLL LLLLLLLLLL
 SOF      HHHHLLLLHH HHLLLLHHHH HHHHLLLLHH HHLLLLHHHH
 Q2       LLHHHHLLLL LLLLLLLLLL LLLLLLLLLL HHHHHHHHHH
 Q1       HHHHHHHHHH LLLLLLLLLL LLHHHHHHHH HHHHHHHHLL
 Q0       HHHHHHHHHH HHHHHHHHHH HHLLLLLLLL LLLLLLLLLL
 BIT      HHHHHHHHHH HHHHHHHHHH LLHHHHHHHH HHHHHHHHHH
 LDB      HHLLLLLLLL LLLLHHHHLL LLLLLLHHHH LLLLHHHHLL
 SUNK     LLLLLLLLLL HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 NO_FRM   LLHHHHLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BYT_CK   LLLLLLLLLL LLLLLLHHHH HHHHLLLLLL LLLLLLLLLL
 BC2      HHHHHHHHHH HHHHHHLLLL LLLLLLLLLL LLLLLLLLLL
 BC1      LLLLLLLLHH HHHHHHLLLL LLLLLLLLLL LLLLLLLLLL
 BC0      LLLLHHHHLL LLHHHHLLLL LLLLHHHHHH HHHHHHHHHH
```

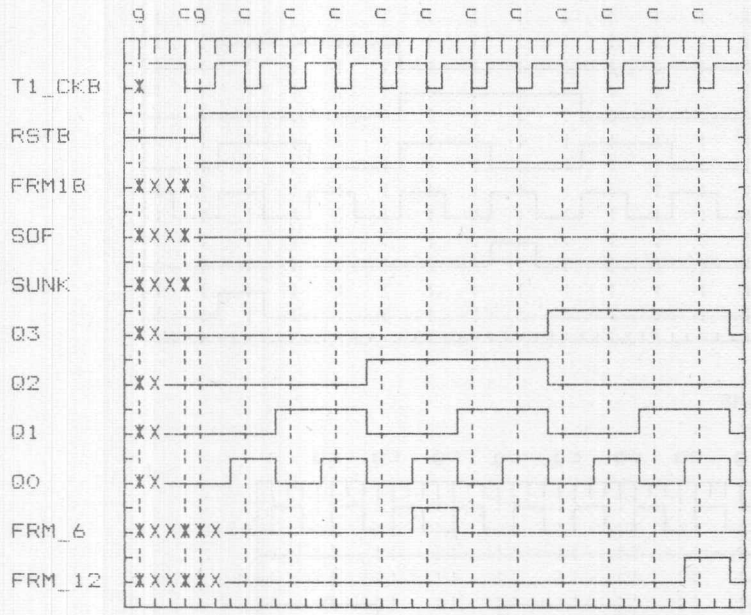
```
SYNC
Page :  3
          g  cg  cg   cg  cg  c   cg  cg   cg  cg  cg
 T1_CKB  LHHLLHHLLH HLLHHLLHHL HHLLHHLLHH LLHHLLHHLL
 RSTB    HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 NO_T1CB HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 DATA    HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 FPD1B   LLLLLLLLLL LLLLLLHHHH HHHLLLLLLL LLLLLLLLLH
 FPD2B   LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LHHHHLLLLL
 SOF     LLLLHHHHLL LLHHHHHHHH HHHLLLLHHH HHHHHHHHHH
 Q2      HHHHHHHHHH HHHHLLLLLL LHHHHLLLLL LLLLLLLHHH
 Q1      LLLLLLLLLL LLLLLLLLHH HHHHHHHHHL LLLHHHHHHH
 Q0      LLLLLLHHHH HHHHHHHHLL LHHHHHHHHH HHHLLLLLLL
 BIT     HHHHHHHHHH HHHHHHLLHH HHHHHHHHHH HLLHHHHHHH
 LDB     LLHHHHLLLL HHHHLLLLLL LLLLLLLLLL LLLLLLLLLL
 SUNK    HHHHHHHHHH HHHHHHHHHH HLLLLLLLLH HHHHHHHHHH
 NO_FRM  LLLLLLLLLL LLLLLLLLLL LHHHHLLLLL LLLLLLLLLL
 BYT_CK  LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC2     LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC1     LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC0     HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH

SYNC
Page :  4
            cg  cg   cg  cg  c   cg  cg  c g  cg  cg
 T1_CKB  HHLLHHLLHH LLHHLLHHLH HLLHHLLHHL LHHLLHHLLH
 RSTB    HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 NO_T1CB HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 DATA    HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 FPD1B   HHHLLLLLLL LLLLLLLLLL LLHHHHHHHH LLLLLLLLLL
 FPD2B   LLLLLLLLLL LHHHHLLLLL LLLLLLLLLL LLLLLLLLHH
 SOF     HHHLLLLHHH HHHHHHHHHH HHLLLLHHHH LLLLHHHHHH
 Q2      HHHHHLLLLL LLLLLLHHHH HHHHHHHHHH HHLLLLLLLL
 Q1      HHHHHHHHHL LLLHHHHHHH LLLLLLLLHH HHHHHLLLLL
 Q0      LHHHHHHHHH HHHLLLLLLL LLLLLLLLHH HHHHHHHHHH
 BIT     HHHHHHHHHH HLLHHHHHHH HHHHHHHHHH HHHHHHHHLL
 LDB     LLLLLLLLLL LLLLLLLLLL LLLLHHHHLL LLLLLLLLLL
 SUNK    HLLLLLLLLH HHHHHHHHHH HHHHHHHHLL LLLLLLHHHH
 NO_FRM  LHHHHLLLLL LLLLLLLLLL LLLLLLLLHH HHLLLLLLLL
 BYT_CK  LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC2     LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC1     LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC0     HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
```

```
SYNC
Page :  5
          cg   c   c    cg   cgg    cg   cg   cg    c   c   c
 T1_CKB   HLLHHLHHLH HLLHHLLLHH LLHHLLHHLL HHLHHLHHLH
 RSTB     HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 NO_T1CB  HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 DATA     HHHHHHHHHH HHHHHHLLLL LLLLLLLLLL LLLLLLLLLL
 FPD1B    LLLLLLLLLL LLHHHHHLLL LLLLLLLLLL LLLLLLLLLL
 FPD2B    HHLLLLLLLL LLLLLLLLLL LLLLLHHHHL LLLLLLLLLL
 SOF      HHHHHHHHHH HHHHHHHLLL LHHHHHHHHH HHHHHHHHHH
 Q2       LLLLHHHHHH HHHHHHHHHL LLLLLLLLLL LHHHHHHHHH
 Q1       HHHHHHHLLL LLLLHHHHHH HHHLLLLHHH HHHHLLLLLL
 Q0       LLLLLLLLLL HHHHHHHHHH HHHHHHLLL LLLLLLHHH
 BIT      HHHHHHHHHH HHHHHHLLLL LLLLLHHLLL LLLLLLLLLL
 LDB      LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 SUNK     HHHHHHHHHH HHHHLLLLLL LLLHHHHHHH HHHHHHHHHH
 NO_FRM   LLLLLLLLLL LLLLHHHHHL LLLLLLLLLL LLLLLLLLLL
 BYT_CK   LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC2      LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC1      LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 BC0      HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH

SYNC
Page :  6
          cg   c
 T1_CKB   HLLHHL
 RSTB     HHHHHH
 NO_T1CB  HHHHHH
 DATA     LLLLLL
 FPD1B    LLHHHH
 FPD2B    LLLLLL
 SOF      HHHHHH
 Q2       LLLLLL
 Q1       LLLLHH
 Q0       HHHHLL
 BIT      LLHHLL
 LDB      LLLLLL
 SUNK     HHHHHH
 NO_FRM   LLLLLL
 BYT_CK   LLLLLL
 BC2      LLLLLL
 BC1      LLLLLL
 BC0      HHHHHH
```

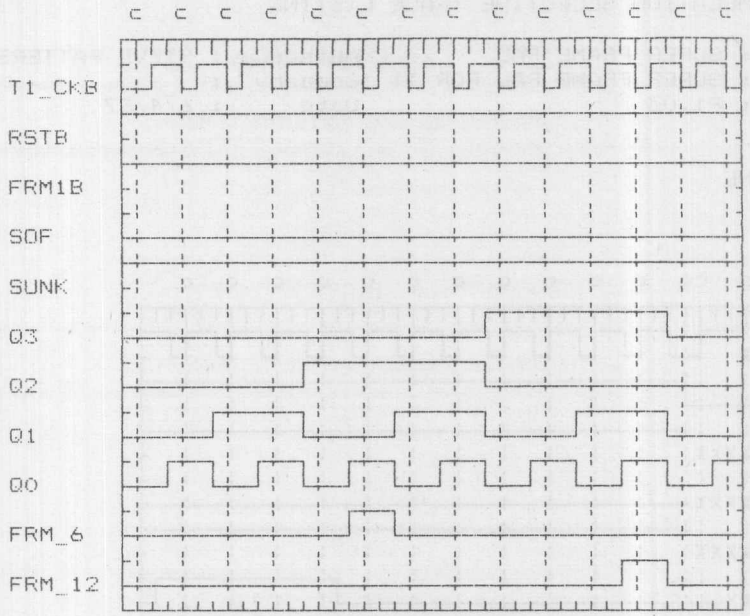
```
TITLE SUPER_FRAME_PAL
PATTERN SUPER FRAME PAL FOR T1 INTERFACE
REVISION P1.03
AUTHOR STEVE PATTERSON AND THERESA SHAFER
COMPANY
DATE 6/4/87

;  This PAL device counts the T1 Frames and controls the Signal Bits
;  extraction process, including Fly Wheeling.  It also provides variou
;  other signals which indicate the frames with signal bits.
;  The counter is reset with either RSTB or when frame detection is SUNK
;  and frame 1 occurs from two different sources (FRM1 & SOF).

CHIP SUPER_FRAME PAL16R6

;PINS
;1       2       3       4       5       6       7       8       9       10
 T1_CKB  RSTB    FRM1B   SUNK    SOF     NC      NC      NC      NC      GND

;11      12      13      14      15      16      17      18      19      20
OEB      NC      Q3      Q2      Q1      Q0      FRM_6   FRM_12  NC      VCC

;INPUTS:T1_CKB          ACTIVE LOW EXTERNAL T1 CLOCK
;       RSTB            ACTIVE LOW MASTER RESET
;       SOF             ACTIVE HIGH INPUT SIGNAL INDICATING
;                         LAST KNOWN START OF FRAME
;       FRM1B           ACTIVE LOW INPUT SIGNAL INDICATING
;                         START OF FRAME 1
;       SUNK            ACTIVE HIGH INPUT SIGNAL INDICATING "IN FRAME SYNC"
;       OEB             ACTIVE LOW OUTPUT ENABLE INPUT

;OUTPUTS:Q(3-0)         STATE VARIABLES
;       FRM_6           CLOCK SIGNAL WHICH INDICATES SIGNAL BIT A
;       FRM_12          CLOCK SIGNAL WHICH INDICATES SIGNAL BIT B


EQUATIONS

/Q3 := /Q2 * Q1 * Q0
    + /Q3 * /Q2
    + /Q3 * /Q1
    + /Q3 * /Q0
    + /FRM1B * SOF * SUNK              ; RESET WHEN SUNK AND FRAME 1 OCCURS
   + /RSTB                             ; MASTER RESET

/Q2 := Q2 * Q1 * Q0
    + /Q2 * Q3
    + /Q2 * /Q1
    + /Q2 * /Q0
    + /FRM1B * SOF * SUNK              ; RESET WHEN SUNK AND FRAME 1 OCCURS
    + /RSTB                            ; MASTER RESET
    + /RSTB                            ; MASTER RESET
```

```
/Q1 := Q1 * Q0
     + /Q1 * /Q0
     + /FRM1B * SOF * SUNK                ; RESET WHEN SUNK AND FRAME 1 OCCURS

/Q0 := Q0
    + /FRM1B * SOF * SUNK                 ; RESET WHEN SUNK AND FRAME 1 OCCURS
    + /RSTB                               ; MASTER RESET


; SIGNAL BIT EXTRACTION OUTPUTS

/FRM_6  := Q3 + /Q2 + Q1 + Q0

/FRM_12 := /Q3 +  Q2 + /Q1 + Q0

SIMULATION

TRACE_ON T1_CKB RSTB FRM1B SOF SUNK
Q3 Q2 Q1 Q0 FRM_6 FRM_12

SETF /OEB                                 ; ENABLE OUTPUT
     /RSTB                                ; RESET REGISTERS
CLOCKF T1_CKB
SETF RSTB /SOF FRM1B SUNK
CLOCKF T1_CKB

FOR I:=1 TO 24 DO                         ; COUNT
        BEGIN
        CLOCKF T1_CKB
        END

SETF /SUNK SOF /FRM1B                     ; RESET CONDITIONS
CLOCKF T1_CKB
SETF /SUNK /SOF /FRM1B
CLOCKF T1_CKB
SETF /SUNK SOF FRM1B
CLOCKF T1_CKB
SETF /SUNK /SOF FRM1B
CLOCKF T1_CKB
SETF SUNK SOF /FRM1B
CLOCKF T1_CKB
SETF SUNK /SOF /FRM1B
CLOCKF T1_CKB
SETF SUNK /SOF FRM1B
CLOCKF T1_CKB
SETF SUNK SOF FRM1B
CLOCKF T1_CKB

TRACE_OFF
```

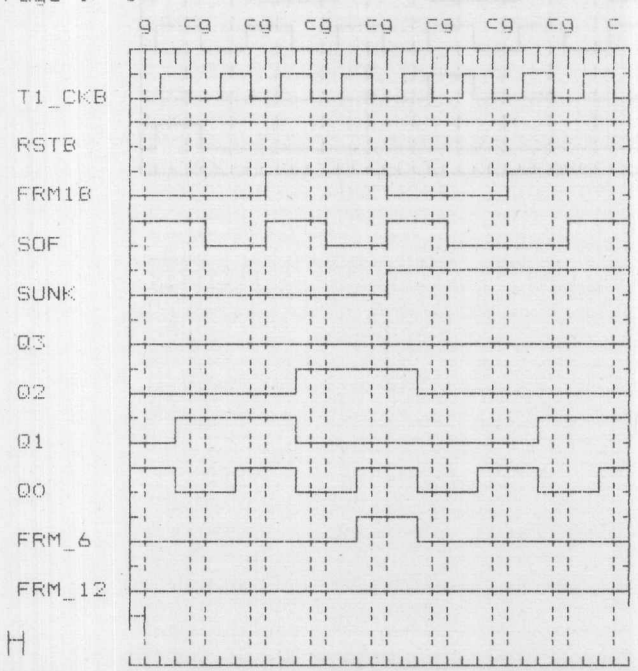
```
PALASM SIMULATION, V2.22 - MARKET RELEASE (11-19-86)
(C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
PALASM SIMULATION SELECTIVE TRACE LISTING

Title    : SUPER_FRAME_PAL          Author   : STEVE PATTERSON AND THER
Pattern  : SUPER FRAME PAL FOR T1 ICompany  :
Revision : P1.03                    Date     : 6/4/87


SUPER_FRAME
Page :  1
```



SUPER_FRAME
Page : 2



SUPER_FRAME
Page : 3



```
  PALASM SIMULATION, V2.22 - MARKET RELEASE (11-19-86)
  (C) - COPYRIGHT MONOLITHIC MEMORIES INC, 1986
  PALASM SIMULATION SELECTIVE TRACE LISTING

  Title    : SUPER_FRAME_PAL          Author   : STEVE PATTERSON AND THER
Pattern  : SUPER FRAME PAL FOR T1 ICompany  :
Revision : P1.03                    Date     : 6/4/87


SUPER_FRAME
Page :  1
         g  cg  c   c  c  c  c   c  c  c   c  c  c
 T1_CKB XHHLLHHLHH LHHLHHLHHL HHLHHLHHLH HLHHLHHLHH
 RSTB   LLLLHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 FRM1B  XXXXHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 SOF    XXXXLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 SUNK   XXXXHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 Q3     XXLLLLLLLL LLLLLLLLLL LLLLLLLHHH HHHHHHHHHL
 Q2     XXLLLLLLLL LLLLLHHHHH HHHHHHHLLL LLLLLLLLLL
 Q1     XXLLLLLLLH HHHHHLLLLL LHHHHHHLLL LLLHHHHHHL
 Q0     XXLLLLHHHL LLHHHLLLHH HLLLHHHLLL HHHLLLHHHL
 FRM_6  XXXXXXLLLL LLLLLLLLHH HLLLLLLLLL LLLLLLLLLL
 FRM_12 XXXXXXLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLHHHL

SUPER_FRAME
Page :  2
         c  c  c  c   c  c  c   c  c  c   c  c  c  c
 T1_CKB LHHLHHLHHL HHLHHLHHLH HLHHLHHLHH LHHLHHLHHL
 RSTB   HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 FRM1B  HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 SOF    LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 SUNK   HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 Q3     LLLLLLLLLL LLLLLLLLLL LLLHHHHHHH HHHHHLLLLL
 Q2     LLLLLLLLLL LHHHHHHHHH HHHLLLLLLL LLLLLLLLLL
 Q1     LLLLLHHHHH HLLLLLLHHH HHHLLLLLLH HHHHHLLLLL
 Q0     LLHHHLLLHH HLLLHHHLLL HHHLLLHHHL LLHHHLLLHH
 FRM_6  LLLLLLLLLL LLLLHHHLLL LLLLLLLLLL LLLLLLLLLL
 FRM_12 LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLHHHLLLLL
```

```
SUPER_FRAME
Page :  3
         g  cg  cg   cg  cg  c g  cg  cg   c
 T1_CKB LHHLLHHLLH HLLHHLLHHL LHHLLHHLLH HL
 RSTB   HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH HH
 FRM1B  LLLLLLLLHH HHHHHHLLLL LLLLHHHHHH HH
 SOF    HHHHLLLLHH HHLLLLHHHH LLLLLLLLHH HH
 SUNK   LLLLLLLLLL LLLLLLHHHH HHHHHHHHHH HH
 Q3     LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LL
 Q2     LLLLLLLLLL HHHHHHHHLL LLLLLLLLLL LL
 Q1     LLHHHHHHHH LLLLLLLLLL LLLLLLHHHH HH
 Q0     HHLLLLHHHH LLLLHHHHLL LLHHHHLLLL HH
 FRM_6  LLLLLLLLLL LLLLHHHHLL LLLLLLLLLL LL
 FRM_12 LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LL
```

T1 Extended Superframe Provides Transmission Error Detection

AN-162

Principles of CRC-6

In digital transmission, bits are grouped into frames. North America's standard, T-carrier uses a framing bit to group data into frames. In T1 extended-superframe (ESF or Fe), the frame-bit position carries additional information besides framing. One such extended functionality provides transmission line error detection. In transmission channels, most errors are bursty in nature; the cyclic-redundancy-checking (CRC) codes provide sufficient detection for these types of errors. With CRC, the block length and the polynomial determine the type and percentage of error which can be detected. For ESF, the block length is one extended-superframe or 4,632 bits. The T1 Fe standard specifies the CRC-6 polynomial. The CRC-6 = $X^6 + X + 1$ polynomial detects 98.4% of one or more errors.

Functional Description

Figure 1 shows a simple configuration of XOR-gates and registers which calculates the CRC-6 polynomial.

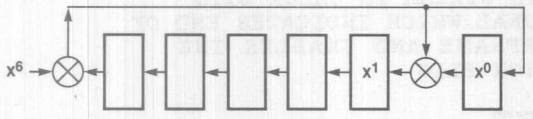


Figure 1. CRC-6 Block Diagram

The low-power PALC16R6Z implements the CRC-6 polynomial for a serial data stream (see Figure 1). At the beginning of CRC generation, the remainder is preset to all zeros by enabling the synchronous CLR. After shifting in the 4,632-bit superframe, the result is the CRC-6 remainder. This remainder is transmitted in the check-bit positions of the subsequent superframe. The remainder can be shifted out serially, most significant bit first or output as a parallel word. The CHECK_BIT signal enables the remainder to be shifted out serially; it must be held HIGH until the entire 6-bit remainder has been shifted out. When CHECK_BIT = 0, CRC6 echoes the serial data input stream. CB1-CB6 are the parallel remainder bits where CB1 is the most significant bit and CB6 is the least significant bit.

There are two implementations for error detection at the receiver. First, the transmitted CRC-6 value is fed serially along with the data. If a non-zero remainder is calculated, an error has occurred. In this case, the ERROR flag indicates at least one transmission error. Second, the CRC-6 for the extended-superframe can be recalculated and externally compared with the transmitted CRC-6 value.

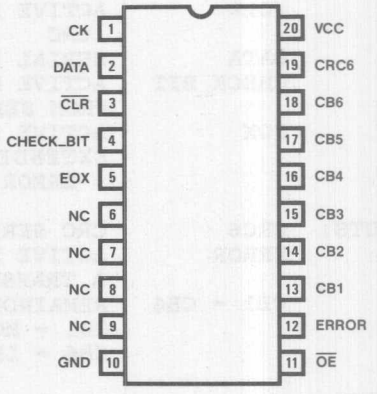


Figure 2. CRC-6 Configuration

PAL® is a registered trademark of Monolithic Memories.
ZPAL™ is a trademark of Monolithic Memories.

PAL® Device Design Specification

```
TITLE CRC6
PATTERN CRC 6 ERROR DETECTION PAL
REVISION 1.01
AUTHOR THERESA SHAFER
COMPANY MMI
DATE 9/10/86

CHIP CRC6 PAL16R6

CK DATA /CLR CHECK_BIT EOX NC NC NC NC GND
/OE ERROR CB1 CB2 CB3 CB4 CB5 CB6 CRC6 VCC

; THE CRC-6 PAL PERFORMS ERROR DETECTION ON A SERIAL DATA
; STREAM.  CRC-6 PAL SUPPORTS THE T1 Fe STANDARD FOR ERROR
; DETECTION. THE CRC RESULT CAN BE OUTPUT EITHER IN SERIAL
; OR IN PARALLEL.
;
;  CRC-6 = X**6 + X + 1
;
; INPUTS:   CK           EXTERNAL CLOCK
;           /OE          ACTIVE LOW OUTPUT ENABLE SIGNAL
;           /CLR         ACTIVE LOW CLEAR SIGNAL WHICH RESETS THE
;                         CRC
;           DATA         SERIAL DATA STREAM INPUT
;           CHECK_BIT    ACTIVE HIGH SELECT INPUT WHICH SWITCHES
;                        FROM SERIAL DATA STREAM TO CHECK BITS
;           EOX          ACTIVE HIGH SIGNAL WHICH INDICATES END OF
;                        EXTENDED SUPERFRAME AND ENABLES THE
;                          ERROR DETECTION FLAG

; OUTPUTS:  CRC6         CRC SERIAL OUTPUT
;           ERROR        ACTIVE HIGH OUTPUT FLAG WHICH INDICATES
;                        A TRANSMISSION ERROR
;           CB1 - CB6    REMAINDER BITS (CHECK BITS)
;                        CB1 - MOST SIGNIFICANT BIT
;                        CB6 - LEAST SIGNIFICANT BIT

EQUATIONS

; CRC-6 PAL

/CRC6 = /CB1 * CHECK_BIT
     + /DATA * /CHECK_BIT

/ERROR = EOX * /CB1 * /CB2 * /CB3 * /CB4 * /CB5 * /CB6
     + /EOX

/CB1 := /CB2 + CLR

/CB2 := /CB3 + CLR

/CB3 := /CB4 + CLR

/CB4 := /CB5 + CLR
```

```
/CB5 := /DATA * /CB1 * /CB6
     + DATA * CB1 * /CB6
     + DATA * /CB1 * CB6 * /CHECK_BIT
     + /DATA * CB1 * CB6 * /CHECK_BIT
     + /CB6 * CHECK_BIT
     + CLR

/CB6 := DATA * CB1 * /CHECK_BIT
     + /DATA * /CB1 * /CHECK_BIT
     + /CB1 * CHECK_BIT
     + CLR

; ..........................................................
; ..........................................................


SIMULATION

TRACE_ON CK /OE /CLR CHECK_BIT EOX DATA CRC6 ERROR
         CB1 CB2 CB3 CB4 CB5 CB6

SETF OE                  ; ENABLE OUTPUT
     CLR                 ; CLEAR SHIFT REGISTER
CLOCKF CK

; INITIALIZE
SETF DATA
     /CHECK_BIT
     /EOX
     /CLR
CLOCKF CK

; FIND PERIOD
SETF /DATA
FOR J:= 0 TO 70 DO
     BEGIN
     CLOCKF CK
     END

; CALULATE CRC-6 FOR 1110010101
SETF CLR                 ; CLEAR SHIFT REGISTER
     DATA
CLOCKF CK
SETF /CLR
     DATA
CLOCKF CK
SETF DATA
CLOCKF CK
SETF DATA
CLOCKF CK
SETF /DATA
CLOCKF CK
SETF /DATA
CLOCKF CK
SETF DATA
```

```
CLOCKF CK
SETF /DATA
CLOCKF CK
SETF DATA
CLOCKF CK
SETF /DATA
CLOCKF CK
SETF DATA
CLOCKF CK

;  CHECK ERROR DETECTION
SETF EOX                    ;  CHECK ERROR FLAG
SETF /DATA                   ;  001001
CLOCKF CK
SETF /DATA
CLOCKF CK
SETF DATA
CLOCKF CK
SETF /DATA
CLOCKF CK
SETF /DATA
CLOCKF CK
SETF DATA
CLOCKF CK

; READ CRC RESULT
SETF /EOX
SETF /DATA
CLOCKF CK
SETF DATA
CLOCKF CK
SETF CHECK_BIT
     DATA
FOR J:= 0 TO 6 DO
     BEGIN
     CLOCKF CK
     END


TRACE_OFF
```

Time Division Multiplexing with the LCA Device

AN-161

Introduction

The Logic Cell Array (LCA device) implements a multiplexer and counter used in time division multiplexing. With the device's flexible I/O pins, the multiplexer and counter are implemented into a single CMOS device. This application note covers time division multiplexing and the design of a multiplexer and counter. Additional information on how to design with an LCA device can be found in the LCA design methodology chapter in Monolithic Memories' LCA Design and Applications Handbook. For programming the LCA device, refer to "Configuring the LCA Device", Monolithic Memories' Application Note 182.

Principles of Multiplexing

Multiplexing efficiently utilizes data communication lines by combining multiple low-speed signals into a single, high-speed line. The two methods of performing multiplexing are Time Division Multiplexing (TDM) and Frequency Division Multiplexing (FDM). TDM divides the transmission bandwidth into equal time slots where each input signal is assigned one time slot per time cycle. Once assigned, that time slot is not used by any other input. Figure 1 shows a multiplexer combining three low-speed signals into one high-speed line. Terminal A transmits during the first time slot, B transmits during the second time slot, and C transmits during the third time slot. This sequence is repeated every time cycle. FDM, on the other hand, divides the frequency spectrum among logical channels where each channel has full bandwidth of its assigned frequencies. Analog systems usually use FDM.

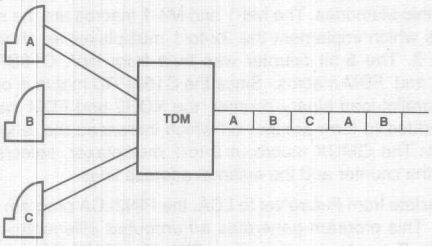


Figure 1. Time Division Multiplexing

There are many classes of TDMs including bit interleave, character interleave, statistical TDM, and T1 multiplexers. In bit interleaved TDM, each input is assigned to one time slot. Each time slot's length is one bit. Character interleaved TDM is similar to bit interleave, except that each time slot represents one character. Statistical multiplexing assigns the bandwidth into unequal slots where only the active incoming lines are assigned slots. The slots are of variable length, so each input is assigned the amount of bandwidth needed. The T1 standard specifies twenty-four 64 kbps channels multiplexed on a 1.544 Mbps high-speed link.

Data Selection/Time Slot Assignment

To transmit data from terminal A in the first time slot as shown in Figure 1, data buffering, rate adaption and data selection must be performed. Data buffering and rate adaption are required since the rates of the lines to be multiplexed are slower than the high-speed link. A serializing FIFO or shift register can implement the required buffering. Rate adaption is performed by a clock/shift scheme tailored to the exact application. Once buffered, the data must be selected in the proper order. Several methods can be used for data selection; one method is a 32-to-1 multiplexer. To implement sequential selection of input lines, an on-board counter selects the multiplexer's output. However, for random selection needed in statistical multiplexing, more flexibility is needed. A 2-to-1 multiplexer provides this flexibility by selecting between the counter and random selection inputs.

Comparison of Design Methodologies

The 32-to-1 multiplexer and counter design can be implemented in several ways. Briefly, we will evaluate discrete logic, PAL and LCA device implementations.

For this design, thirty-eight inputs, five registers, and one output are required. Conventional PLD devices do not meet this large I/O requirement. A single PLD device is not suited for this specific application because flexible I/O and buried registers are not supported. Instead, the design must be divided into four sections. The counter fits in a PALC16R6Z device and the 32-to-1 multiplexer requires three PALC20L8Z devices.

Since a large 32-to-1 multiplexer is not available as a discrete part, it must be built using smaller multiplexers. The total chip count, using discrete logic, is five devices: two 74AS850s, two 74ALS257s and one 74AS867. A combination of PAL devices and discrete logic devices is possible, however, it is also a multiple chip solution.

Monolithic Memories' LCA device is a high-density programmable CMOS circuit with flexible I/Os. It has forty pins which are user-defined as either inputs, outputs, or bidirectional. The LCA device meets the I/O requirement for this design. Since the LCA circuit allows multiple logic levels, implementing the counter does not use up output pins. Moreover an LCA device enables the 32-to-1 multiplexer with counter to be implemented in a single low-power device.

Detailed LCA Design

The M2064 LCA device in a 48-pin DIP is used in this design, although both PLCC and PGA packages are available. Eight of the forty-eight pins are reserved for programming, power, and ground, and the remaining forty I/O pins are available to the user. Table 1 shows the efficient use of the package pinout.

PIN	DESCRIPTION	TOTAL PINS
ADDR(4:0)	External Address Inputs	5 inputs
CK	External Clock Input	1 input
LD	Counter Load Input	1 input
D(31:0)	32-to-1 Multiplexer Data Inputs.	32 inputs
OUT	Output	1 output
	TOTAL I/O	40 pins

Table 1. 32-to-1 Multiplexer Pins

Figure 2 shows the block diagram of the 32-to-1 multiplexer and the counter. The select lines of the 32-to-1 multiplexer are either the address lines or the output of the counter. When LD is deasserted, the 5-bit counter sequentially selects each input. Asserting the LD signal loads the counter with the external addresses.It also selects the external address for the 32-to-1 multiplexer's select lines. This permits random input selection from the external address lines which supports statistical multiplexing.

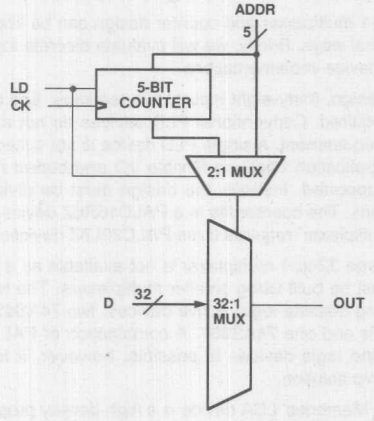


Figure 2. Block Diagram

The select signals for the 32-to-1 multiplexer are determined by the LD signal. When LD = 0, the counter selects the multiplexer's output. When LD = 1, the counter is loaded and the external address selects the output. Also, the LD asynchronously loads the 5-bit counter. Table 2 summarizes the device operation.

INPUTS	OUTPUT	FUNCTION
LD ADDR(4:0)	OUT	
0 XXXXX	D(CNT)	Counter as Select
1 ADDR(4:0)	D(ADDR)	Address as Select and Load Counter

Table 2. 32-to-1 Multiplexer Function Table

The design was entered with FutureNet's DASH schematic capture package using high-level macros. Then the schematic capture file was translated into an LCA design file. The translation software, called PIN2LCA, partitions the design into CLBs and generates the equations for the CLBs' configuration. The translation software produces an unrouted LCA design file. This design file requires final placement of CLBs and interconnect routing which was performed using Monolithic Memories' APR software, an automatic place and route program. After the design is placed and routed, it was optimized with Monolithic Memories' XACT Design Editor System. The XACT system provides a graphic interface to manually optimize the CLB logic functions and connections.

The logic functions and interconnects of an LCA device are established with CMOS memory cells, so the array is never physically altered, although it is physically programmed. A programmable LCA device can be reconfigured for the prototype. In addition, it can be reprogrammed any number of times in the target system. With reconfigurable devices such as this one, the array can also be programmed on power-up, or whenever the design details need to be changed. For volume production, a hard-array version will be available.

Entering the Design with Futurenet

The schematic entered in FutureNet's DASH is comprised of different components called from the macro library supplied by Monolithic Memories. The M8-1 and M4-1 macros are the multiplexers which implement the 32-to-1 multiplexer as shown in Figure 3. The 5-bit counter was built from INV, C16BPRD, XOR2, and FDM macros. Since the C16BPRD macro is only a 4-bit parallel-load binary counter, the XOR2 and FDM macros were added to form another bit which increased the length to five bits.The GMUX macro, a 2-to-1 multiplexer, selects between the counter and the external address lines.

To translate from FutureNet to LCA, the PIN2LCA program is invoked. This program generates an unrouted <filename>.LCA from the FutureNet's <filename>.PIN. The PIN2LCA software partitions the design into CLBs and generates each CLB's configuration.

The PIN2LCA performs logic reduction by eliminating unused logic to maximize CLB utilization. With this design, for example, C16BPRD's reset logic is not used, and PIN2LCA eliminates all the reset logic in the C16BPRD macro. Another means to maximize CLB utilization is by combining macro logic. At times, this combined logic results in a single CLB which implements logic from two different macros. For designs not utilizing all the CLBs, this logic reduction is not required and may actually hinder placement and delay tradeoffs. If it does produce delay problems, it is possible to edit the CLBs in XACT.

At this stage in the design flow, the PIN2LCA software can generate a P-SILOS simulation file. Since the <filename>.SIM was generated from an unrouted LCA design file, the simulation only has unit delays and does not have actual routing delays. The simulation verifies the logic, but does not give any timing information. It may be necessary to make changes in DASH™ depending on the simulation results.

Place and Route Using APR

Once the logic is finalized, the LCA design must be placed and routed. Monolithic Memories' APR software performs the final placement of CLBs and interconnect routing. The input to APR is an unrouted <filename>.LCA, and a routed LCA design file and report document are the outputs.

At first glance, the APR program seems cumbersome to apply to every application. However, it can be used to create an efficient design. While the software uses random placement, it does allow tailoring for specific requirements.

Initially, the multiplexer and counter design did not route completely. To achieve good placement and 100% routing completion, several things were tried. Some worked well while others did not. An overview of how 100% routing completion was achieved is explained below.

Since this design utilizes only 53% of the LCA device, delay is a larger concern than CLB utilization. In this case, placement is critical to achieve an efficient design with the desired delays. The ideal placement would be for the 5-bit counter and 2-to-1 multiplexer to be placed near the ADDR(4:0), CK, and LD signals. The four M8-1 macros should be near the 8-bit cluster of inputs. The remaining M4-1 macro should be placed near the OUT signal.

In FutureNet, the I/Os are assigned pin numbers so that the signals are clustered into groups. These I/O assignments restrict the APR's placement of the signals and are found in the <filename>.SCP. Other restrictions and options are specified when invoking the APR software. The options, "-a", "-g", "-e", and "-k", tailor the placement for this specific design.

The APR's "-a" option specifies the depth of CLB logic levels which are considered connected. In this design, "-a2" was selected because the C16BPRD and M8-1 macros are implemented in two CLB logic levels.

The "-g" and "-e" options are related. The "-g" option specifies the number of CLBs that should be grouped together. The "-e" option determines the number of CLBs in a group which should be evaluated for all possible placements. Since the C16BPRD and M8-1 macros contain four to six CLBs, the options, "-e4" and "-g4", were used. This particular value was selected because it was the largest possible value which maintained the e>=g relationship. Avoiding values where e<g insures as much as possible that the CLBs within each group are placed in the best possible arrangement. This maintains the groups' integrity.

The "-k" option specifies the number of shapes per group. By trial and error, "-k6" works well for this design. With "-k6", six of the best arrangements or shapes were saved for each group of CLBs. Every combination of shaped group is tried with all six other grouped shapes and evaluated for best placement. Generally, the larger values result in better placements, however, run-time grows exponentially. While larger values of "-k" were tried, they did not improve the placement significantly.

Running APR with these options and restrictions did not result in a 100% routed design. Therefore, more application-specific information was required. Noting that the APR software was not fully utilizing the high-level macro information, placement could still be optimized. The APR's report file shows that the CLBs which comprise C16BPRD or the M8-1 macros were not grouped together. Instead, the CLBs were randomly spread over the device. In a constraint file, <filename>.CST, CLBs from the high-level macros could be grouped together (see Appendix A). If necessary, the exact CLB placement could be specified in the same file.

To generate the constraint file, CLBs which implement each high-level macro were identified using the cross-reference file generated by PIN2LCA, <filename>.CRF. The <filename>.CRF file is organized into three cross-reference sections: macros, CLBs, and IOBs.

The macro cross-reference section identifies and assigns a hierarchical symbol number to each macro. The symbol number is used to generate the default CLB names.

```
                    21-1, \M8-1.DWG
                                Path:
ASSIGNED
SYMBOL NUMBER                   \USER\THERESA\TDM.DWG(32)
                    Title: M8-1
```

The CLB cross-reference section of the <filename>.CRF contains the CLB names. For macros with multiple CLBs, the system assigns default names. The default CLB name consists of two parts. The first is the macro symbol number as specified in the macro section and the second number is the signal name. By grouping CLBs with related symbol numbers, all the CLBs for a given macro will be placed together.

```
CLB BD Name = '21-D03':
    F = signal '21-D03', contains:
        symbol '26-OR(2)', output signal = '21-D03'
        symbol '26-AND(3)', output signal = '26-S0'
        symbol '27-OR(2)', output signal = '21-D01'
        symbol '26-AND)1)', output signal = '26-S1'
```

The IOB cross-reference section shows the IOB assignments. For this design, the assignments are those specified in the FutureNet's schematic.

```
IOB P30: Name = 'D13', Symbol = 'PIN(40)'
    I = signal 'D13'
```

Design Editor System

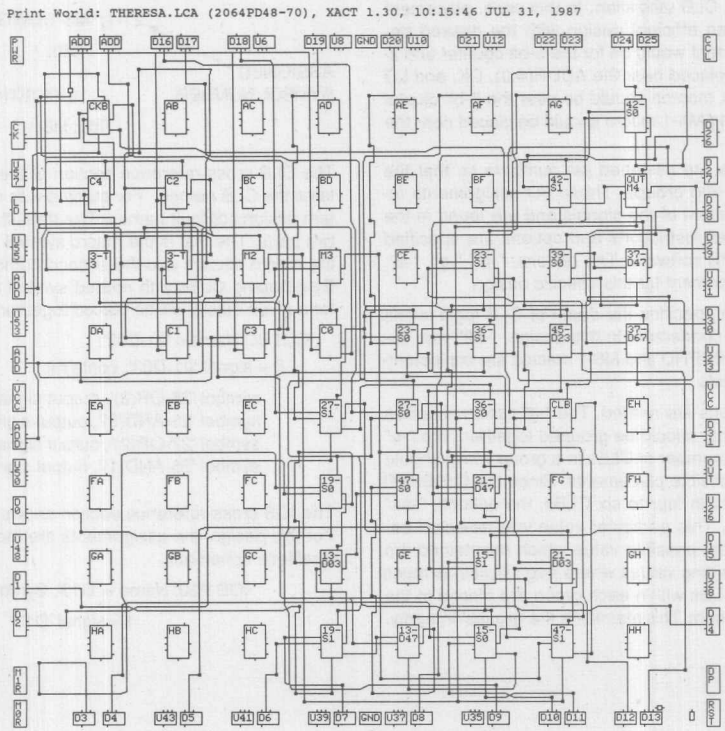
Monolithic Memories' XACT Design Editor System can increase resource utilization and performance by manually modifying the placement and routing. The XACT system provides a graphic interface to specify the CLB design. All CLB logic functions and connections can be optimized for the designs' needs. With XACT, the designer can partition the design and optimize the placement of logic blocks to gain the utilization and performance required.

The clock line routing was optimized to reduce clock skew. Using the clock buffer resources, the common clock is driven by the global clock buffer. To optimize internal routing, "long lines" were used for signals with large fanouts such as the select lines for the multiplexers and the LD signal. Use of the "long lines" minimizes the delay for these control signals. The "long lines" were selected by manually routing the signal net through the programmable interconnect points (PIPs).

On the device itself, implementing the 5-bit counter and 2-to-1 multiplexers requires ten CLBs; and the 32-to-1 multiplexer requires twenty-four CLBs, giving a total of thirty-four CLBs. This design utilizes 53% of the LCA device.

implemented in a single CMOS device. FutureNet's DASH schematic capture with Monolithic Memories' LCA macro library was used to implement a design in an LCA device. Placement of CLBs and signal routing were performed by Monolithic Memories' APR software. Optimization, which may be required to improve performance, can be done using Monolithic Memories' XACT Design Editor System. This is useful for placing and routing critical signal nets such as clock or control signals. More information can be found in P-SILOS, FutureNet, and LCA user guides and handbooks.

This design, developed with an LCA device, is available upon request. The FutureNet drawing, LCA design and bit pattern files can be provided for programming the LCA device in an EPROM. Please ask for design XDES07.LCA.



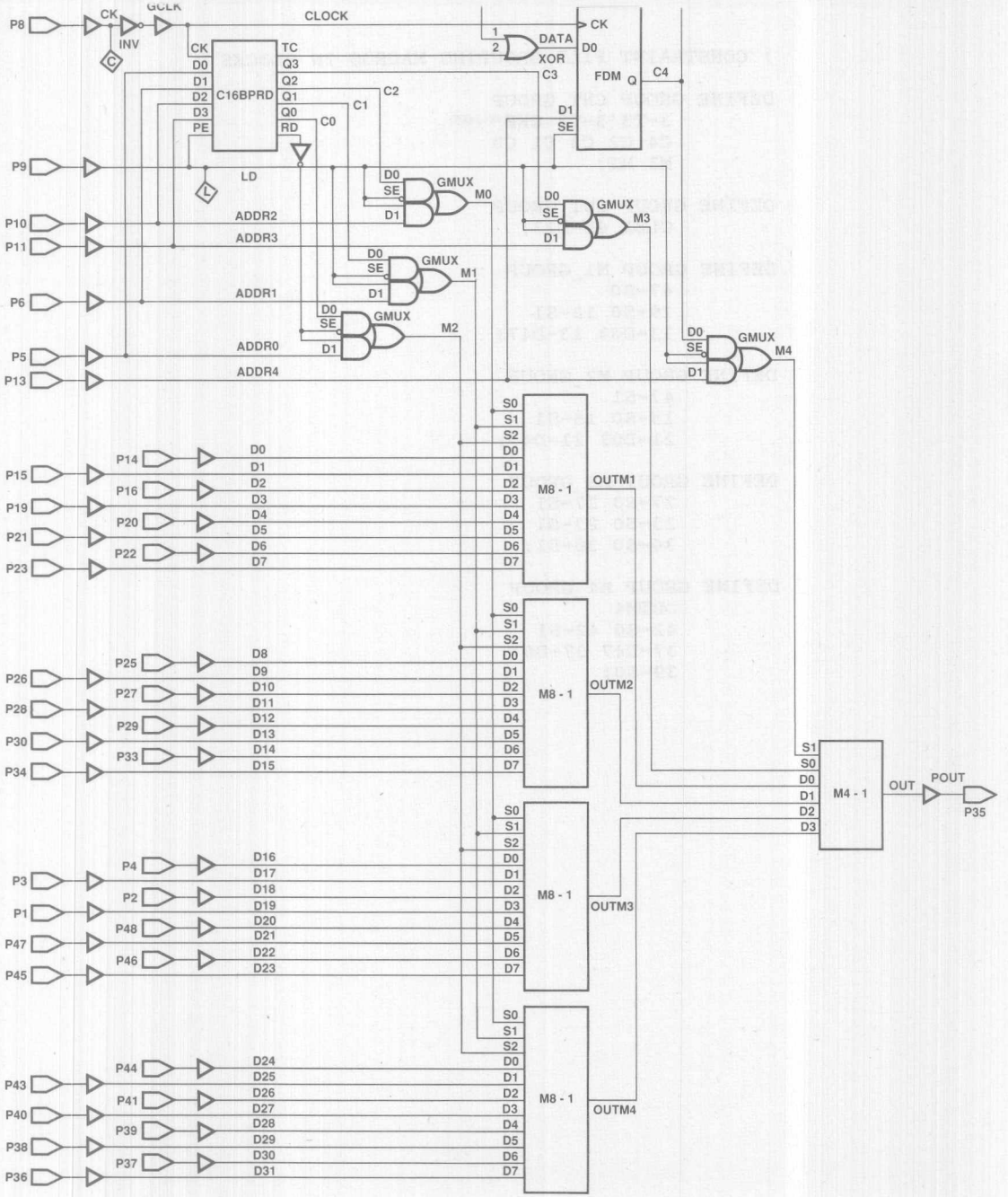


Figure 3. FutureNet Schematic

Appendix A> Constraint File

```
; CONSTRAINT FILE GROUPING MACROS IN BLOCKS

DEFINE GROUP CNT_GROUP
        3-T3 3-T2 CKB
        C4 C2 C3 C1 C0
        M3 M2;

DEFINE GROUP OUT_GROUP
        CLB1 45-D23;

DEFINE GROUP M1_GROUP
        47-S0
        19-S0 19-S1
        13-D03 13-D47;

DEFINE GROUP M2_GROUP
        47-S1
        15-S0 15-S1
        21-D03 21-D47;

DEFINE GROUP M3_GROUP
        27-S0 27-S1
        23-S0 23-S1
        36-S0 36-S1;

DEFINE GROUP M4_GROUP
        OUTM4
        42-S0 42-S1
        37-D47 37-D67
        39-S0;
```

```
$
$ Simulation file for design 'TDM.sim', type '2064N48-50'
$ Created by PIN2LCA Ver. 1.01x at 10:58:00 JUL 14, 1987
$
!INPUT TDM.sim

GLOBALRESET-      .CLK 0 S0 1 S1 $ Initial pulse to reset latches
CK.PAD            .CLK 0 0 500 1 1000 0 .REP 0
LD.PAD            .CLK 0 0 40000 1
ADDR0.PAD         .CLK 0 0 40000 0 41000 1 42000 0 .REP 40000
+ 78000 0
ADDR1.PAD         .CLK 0 0 40000 0 42000 1 44000 0 .REP 40000
+ 78000 1
ADDR2.PAD         .CLK 0 0 40000 0 44000 1 48000 0 .REP 40000
+ 78000 1
ADDR3.PAD         .CLK 0 0 40000 0 48000 1 56000 0 .REP 40000
+ 78000 0
ADDR4.PAD         .CLK 0 0 40000 0 56000 1 72000 0 .REP 40000
+ 78000 1
D0.PAD            .CLK 0 1  1000 0          40000 0 41000 1
D1.PAD            .CLK 0 0  1000 1  2000 0 40000 1 41000 0 42000 1
D2.PAD            .CLK 0 0  2000 1  3000 0 40000 1 42000 0 43000 1
D3.PAD            .CLK 0 0  3000 1  4000 0 40000 1 43000 0 44000 1
D4.PAD            .CLK 0 0  4000 1  5000 0 40000 1 44000 0 45000 1
D5.PAD            .CLK 0 0  5000 1  6000 0 40000 1 45000 0 46000 1
D6.PAD            .CLK 0 0  6000 1  7000 0 40000 1 46000 0 47000 1
D7.PAD            .CLK 0 0  7000 1  8000 0 40000 1 47000 0 48000 1
D8.PAD            .CLK 0 0  8000 1  9000 0 40000 1 48000 0 49000 1
D9.PAD            .CLK 0 0  9000 1 10000 0 40000 1 49000 0 50000 1
D10.PAD           .CLK 0 0 10000 1 11000 0 40000 1 50000 0 51000 1
D11.PAD           .CLK 0 0 11000 1 12000 0 40000 1 51000 0 52000 1
D12.PAD           .CLK 0 0 12000 1 13000 0 40000 1 52000 0 53000 1
D13.PAD           .CLK 0 0 13000 1 14000 0 40000 1 53000 0 54000 1
D14.PAD           .CLK 0 0 14000 1 15000 0 40000 1 54000 0 55000 1
D15.PAD           .CLK 0 0 15000 1 16000 0 40000 1 55000 0 56000 1
D16.PAD           .CLK 0 0 16000 1 17000 0 40000 1 56000 0 57000 1
D17.PAD           .CLK 0 0 17000 1 18000 0 40000 1 57000 0 58000 1
D18.PAD           .CLK 0 0 18000 1 19000 0 40000 1 58000 0 59000 1
D19.PAD           .CLK 0 0 19000 1 20000 0 40000 1 59000 0 60000 1
D20.PAD           .CLK 0 0 20000 1 21000 0 40000 1 60000 0 61000 1
D21.PAD           .CLK 0 0 21000 1 22000 0 40000 1 61000 0 62000 1
D22.PAD           .CLK 0 0 22000 1 23000 0 40000 1 62000 0 63000 1 73000 0
D23.PAD           .CLK 0 0 23000 1 24000 0 40000 1 63000 0 64000 1
D24.PAD           .CLK 0 0 24000 1 25000 0 40000 1 64000 0 65000 1
D25.PAD           .CLK 0 0 25000 1 26000 0 40000 1 65000 0 66000 1
D26.PAD           .CLK 0 0 26000 1 27000 0 40000 1 66000 0 67000 1
D27.PAD           .CLK 0 0 27000 1 28000 0 40000 1 67000 0 68000 1
D28.PAD           .CLK 0 0 28000 1 29000 0 40000 1 68000 0 69000 1
D29.PAD           .CLK 0 0 29000 1 30000 0 40000 1 69000 0 70000 1
D30.PAD           .CLK 0 0 30000 1 31000 0 40000 1 70000 0 71000 1
D31.PAD           .CLK 0 0 31000 1 32000 0 40000 1 71000 0 72000 1

 .MONITOR CK.PAD LD.PAD ; ADDR4.PAD ADDR3.PAD ADDR2.PAD ADDR1.PAD ADDR0.PAD ; ;
+ C4 C3 C2 C1 C0 ;
+ D0.PAD D1.PAD D2.PAD D3.PAD D4.PAD D5.PAD D6.PAD D7.PAD ;
+ D8.PAD D9.PAD D10.PAD D11.PAD D12.PAD D13.PAD D14.PAD D15.PAD ;
+ D16.PAD D17.PAD D18.PAD D19.PAD D20.PAD D21.PAD D22.PAD D23.PAD ;
+ D24.PAD D25.PAD D26.PAD D27.PAD D28.PAD D29.PAD D30.PAD D31.PAD ; OUT.PAD

 .TABLE CK.PAD LD.PAD ; ADDR4.PAD ADDR3.PAD ADDR2.PAD ADDR1.PAD ADDR0.PAD ; ;
+ C4 C3 C2 C1 C0 ;
+ D0.PAD D1.PAD D2.PAD D3.PAD D4.PAD D5.PAD D6.PAD D7.PAD ;
+ D8.PAD D9.PAD D10.PAD D11.PAD D12.PAD D13.PAD D14.PAD D15.PAD ;
+ D16.PAD D17.PAD D18.PAD D19.PAD D20.PAD D21.PAD D22.PAD D23.PAD ;
+ D24.PAD D25.PAD D26.PAD D27.PAD D28.PAD D29.PAD D30.PAD D31.PAD ; OUT.PAD
```

	.. DDDDD		.....	..012345	67890123	45678901	T
	PP RRRRR		PPPPPPPP	PP.....	.....	.....	.
	AA 43210		AAAAAAAA	AAPPPPPP	PPPPPPPP	PPPPPPPP	P
	DD		DDDDDDDD	DDAAAAAA	AAAAAAAA	AAAAAAAA	A
	PPPPP			DDDDDD	DDDDDDDD	DDDDDDDD	D
	AAAAA						
	DDDDD						
TIME							
0	00 00000	00000	10000000	00000000	00000000	00000000	1
500	10 00000	00000	10000000	00000000	00000000	00000000	1
1000	00 00000	00000	01000000	00000000	00000000	00000000	1
1500	10 00000	00001	01000000	00000000	00000000	00000000	1
2000	00 00000	00001	00100000	00000000	00000000	00000000	1
2500	10 00000	00010	00100000	00000000	00000000	00000000	1
3000	00 00000	00010	00010000	00000000	00000000	00000000	1
3500	10 00000	00011	00010000	00000000	00000000	00000000	1
4000	00 00000	00011	00001000	00000000	00000000	00000000	1
4500	10 00000	00100	00001000	00000000	00000000	00000000	1
5000	00 00000	00100	00000100	00000000	00000000	00000000	1
5500	10 00000	00101	00000100	00000000	00000000	00000000	1
6000	00 00000	00101	00000010	00000000	00000000	00000000	1
6500	10 00000	00110	00000010	00000000	00000000	00000000	1
7000	00 00000	00110	00000001	00000000	00000000	00000000	1
7500	10 00000	00111	00000001	00000000	00000000	00000000	1
8000	00 00000	00111	00000000	10000000	00000000	00000000	1
8500	10 00000	01000	00000000	10000000	00000000	00000000	1
9000	00 00000	01000	00000000	01000000	00000000	00000000	1
9500	10 00000	01001	00000000	01000000	00000000	00000000	1
10000	00 00000	01001	00000000	00100000	00000000	00000000	1
10500	10 00000	01010	00000000	00100000	00000000	00000000	1
11000	00 00000	01010	00000000	00010000	00000000	00000000	1
11500	10 00000	01011	00000000	00010000	00000000	00000000	1
12000	00 00000	01011	00000000	00001000	00000000	00000000	1
12500	10 00000	01100	00000000	00001000	00000000	00000000	1
13000	00 00000	01100	00000000	00000100	00000000	00000000	1
13500	10 00000	01101	00000000	00000100	00000000	00000000	1
14000	00 00000	01101	00000000	00000010	00000000	00000000	1
14500	10 00000	01110	00000000	00000010	00000000	00000000	1
15000	00 00000	01110	00000000	00000001	00000000	00000000	1
15500	10 00000	01111	00000000	00000001	00000000	00000000	1
16000	00 00000	01111	00000000	00000000	10000000	00000000	1
16500	10 00000	10000	00000000	00000000	10000000	00000000	1
17000	00 00000	10000	00000000	00000000	01000000	00000000	1
17500	10 00000	10001	00000000	00000000	01000000	00000000	1
18000	00 00000	10001	00000000	00000000	00100000	00000000	1
18500	10 00000	10010	00000000	00000000	00100000	00000000	1
19000	00 00000	10010	00000000	00000000	00010000	00000000	1
19500	10 00000	10011	00000000	00000000	00010000	00000000	1
20000	00 00000	10011	00000000	00000000	00001000	00000000	1
20500	00 00000	10011	00000000	00000000	00001000	00000000	1
21000	00 00000	10100	00000000	00000000	00000100	00000000	1
21500	10 00000	10101	00000000	00000000	00000100	00000000	1
22000	00 00000	10101	00000000	00000000	00000010	00000000	1
22500	10 00000	10110	00000000	00000000	00000010	00000000	1
23000	00 00000	10110	00000000	00000000	00000001	00000000	1
23500	10 00000	10111	00000000	00000000	00000001	00000000	1
24000	00 00000	10111	00000000	00000000	00000000	10000000	1
24500	10 00000	11000	00000000	00000000	00000000	10000000	1
25000	00 00000	11000	00000000	00000000	00000000	01000000	1
25500	10 00000	11001	00000000	00000000	00000000	01000000	1
26000	00 00000	11001	00000000	00000000	00000000	00100000	1
26500	10 00000	11010	00000000	00000000	00000000	00100000	1
27000	00 00000	11010	00000000	00000000	00000000	00010000	1
27500	10 00000	11011	00000000	00000000	00000000	00010000	1
28000	00 00000	11011	00000000	00000000	00000000	00001000	1
28500	10 00000	11100	00000000	00000000	00000000	00001000	1
29000	00 00000	11100	00000000	00000000	00000000	00000100	1
29500	10 00000	11101	00000000	00000000	00000000	00000100	1
30000	00 00000	11101	00000000	00000000	00000000	00000010	1
30500	10 00000	11110	00000000	00000000	00000000	00000010	1
31000	00 00000	11110	00000000	00000000	00000000	00000001	1
31500	10 00000	11111	00000000	00000000	00000000	00000001	1
32000	00 00000	11111	00000000	00000000	00000000	00000000	1
32500	10 00000	00000	00000000	00000000	00000000	00000000	0
33000	00 00000	00000	00000000	00000000	00000000	00000000	0
39000	00 00000	00110	00000000	00000000	00000000	00000000	0
39500	10 00000	00111	00000000	00000000	00000000	00000000	0
40000	01 00000	00111	01111111	11111111	11111111	11111111	0
40500	11 00000	01000	01111111	11111111	11111111	11111111	0
41000	01 00001	01000	10111111	11111111	11111111	11111111	0
41500	11 00001	00001	10111111	11111111	11111111	11111111	0
42000	01 00010	00001	11011111	11111111	11111111	11111111	0
42500	11 00010	00010	11011111	11111111	11111111	11111111	0

TIME	CL KD .. PP AA DD PPPPP AAAAA DDDDD	AAAAA DDDDD ..... RRRRR 43210	CCCCC 43210	DDDDDDDD 01234567 ..... PPPPPPPP AAAAAAAA DDDDDDDD	DDDDDDDD 89111111 ..012345 PP..... AAPPPPPP DDAAAAAA DDDDDD	DDDDDDDD 11112222 67890123 ..... PPPPPPPP AAAAAAAA DDDDDDDD	DDDDDDDD 22222233 45678901 ..... PPPPPPPP AAAAAAAA DDDDDDDD	O U T . P A D
43000	01	00011	00010	11101111	11111111	11111111	11111111	0
43500	11	00011	00011	11101111	11111111	11111111	11111111	0
44000	01	00100	00011	11110111	11111111	11111111	11111111	0
44500	11	00100	00000	11110111	11111111	11111111	11111111	0
45000	01	00101	00000	11111011	11111111	11111111	11111111	0
45500	11	00101	00101	11111011	11111111	11111111	11111111	0
46000	01	00110	00101	11111101	11111111	11111111	11111111	0
46500	11	00110	00110	11111101	11111111	11111111	11111111	0
47000	01	00111	00110	11111110	11111111	11111111	11111111	0
47500	11	00111	00111	11111110	11111111	11111111	11111111	0
48000	01	01000	00111	11111111	01111111	11111111	11111111	0
48500	11	01000	01100	11111111	01111111	11111111	11111111	0
49000	01	01001	01100	11111111	10111111	11111111	11111111	0
49500	11	01001	01001	11111111	10111111	11111111	11111111	0
50000	01	01010	01001	11111111	11011111	11111111	11111111	0
50500	11	01010	01010	11111111	11011111	11111111	11111111	0
51000	01	01011	01010	11111111	11101111	11111111	11111111	0
51500	11	01011	01011	11111111	11101111	11111111	11111111	0
52000	01	01100	01011	11111111	11110111	11111111	11111111	0
52500	11	01100	01000	11111111	11110111	11111111	11111111	0
53000	01	01101	01000	11111111	11111011	11111111	11111111	0
53500	11	01101	01101	11111111	11111011	11111111	11111111	0
54000	01	01110	01101	11111111	11111101	11111111	11111111	0
54500	11	01110	01110	11111111	11111101	11111111	11111111	0
55000	01	01111	01110	11111111	11111110	11111111	11111111	0
55500	11	01111	01111	11111111	11111110	11111111	11111111	0
56000	01	10000	01111	11111111	11111111	01111111	11111111	0
56500	11	10000	10100	11111111	11111111	01111111	11111111	0
57000	01	10001	10100	11111111	11111111	10111111	11111111	0
57500	11	10001	10001	11111111	11111111	10111111	11111111	0
58000	01	10010	10001	11111111	11111111	11011111	11111111	0
58500	11	10010	10010	11111111	11111111	11011111	11111111	0
59000	01	10011	10010	11111111	11111111	11101111	11111111	0
59500	11	10011	10011	11111111	11111111	11101111	11111111	0
60000	01	10100	10011	11111111	11111111	11110111	11111111	0
60500	11	10100	10000	11111111	11111111	11110111	11111111	0
61000	01	10101	10000	11111111	11111111	11111011	11111111	0
61500	11	10101	10101	11111111	11111111	11111011	11111111	0
62000	01	10110	10101	11111111	11111111	11111101	11111111	0
62500	11	10110	10110	11111111	11111111	11111101	11111111	0
63000	01	10111	10110	11111111	11111111	11111110	11111111	0
63500	11	10111	10111	11111111	11111111	11111110	11111111	0
64000	01	11000	10111	11111111	11111111	11111111	01111111	0
64500	11	11000	11100	11111111	11111111	11111111	01111111	0
65000	01	11001	11100	11111111	11111111	11111111	10111111	0
65500	11	11001	11001	11111111	11111111	11111111	10111111	0
66000	01	11010	11001	11111111	11111111	11111111	11011111	0
66500	11	11010	11010	11111111	11111111	11111111	11011111	0
67000	01	11011	11010	11111111	11111111	11111111	11101111	0
67500	11	11011	11011	11111111	11111111	11111111	11101111	0
68000	01	11100	11011	11111111	11111111	11111111	11110111	0
68500	11	11100	11000	11111111	11111111	11111111	11110111	0
69000	01	11101	11000	11111111	11111111	11111111	11111011	0
69500	11	11101	11101	11111111	11111111	11111111	11111011	0
70000	01	11110	11101	11111111	11111111	11111111	11111101	0
70500	11	11110	11110	11111111	11111111	11111111	11111101	0
71000	01	11111	11110	11111111	11111111	11111111	11111110	0
71500	11	11111	11111	11111111	11111111	11111111	11111110	0
72000	01	00000	11111	11111111	11111111	11111111	11111111	0
72500	11	00000	00100	11111111	11111111	11111111	11111111	1
73000	01	00001	00100	11111111	11111111	11111101	11111111	1
73500	11	00001	00001	11111111	11111111	11111101	11111111	1
74000	01	00010	00001	11111111	11111111	11111101	11111111	1
74500	11	00010	00010	11111111	11111111	11111101	11111111	1
75000	01	00011	00010	11111111	11111111	11111101	11111111	1
75500	11	00011	00011	11111111	11111111	11111101	11111111	1
76000	01	00100	00011	11111111	11111111	11111101	11111111	1
76500	11	00100	00000	11111111	11111111	11111101	11111111	1
77000	01	00101	00000	11111111	11111111	11111101	11111111	1
77500	11	00101	00101	11111111	11111111	11111101	11111111	1
78000	01	10110	00101	11111111	11111111	11111101	11111111	1
78500	11	10110	10110	11111111	11111111	11111101	11111111	0
79000	01	10110	10110	11111111	11111111	11111101	11111111	0
79500	11	10110	10110	11111111	11111111	11111101	11111111	0
80000	01	10110	10110	11111111	11111111	11111101	11111111	0

LCA Device Implements an 8-Bit Format Converter in a PBX Switching Module

AN-178

Introduction

In a Private Branch Exchange (PBX) or Central Office (CO), incoming digitized voice circuits are "switched" or routed to their appropriate destination. The "switching" takes place while the data is in serial or parallel format. An eight-bit format converter circuit, widely used in a variety of PBX and CO architectures, translates the serial data streams into parallel data streams and vice-versa.

The eight-bit format converter circuit requires a large number of register elements. These elements, namely serial shift registers, can be implemented efficiently in an LCA device due to its high register content and flexible structure. The LCA M2018 programmable device contains one hundred configurable register elements, ninety-nine of which are used in the implementation of the converter circuit. This circuit is capable of running at 12.5 MHz and 18.5 MHz in a 50-MHz and 70-MHz LCA device, respectively.

Powerful software tools and circuit uniformity allowed the design to be laid out and simulated in just two days. However, if different support logic becomes necessary, PBX vendors are able to generate a new configuration program in the LCA reprogrammable device. Since these devices are manufactured in a CMOS technology, a complete PBX system with several LCA devices can help keep the active power consumption down. (Please refer to the LCA Design and Applications Handbook-Reference 3 for basic information regarding structure and programming aspects.)

Overview of a PBX

Central Offices (CO) and Private Branch Exchanges (PBX) provide both telephone and data services. A PBX may be considered as a localized telephone exchange serving the interconnection requirements of users in office environments. Outgoing calls are directed out of the PBX and routed through the local CO to the far destination. In general terms, a PBX or a CO consists of major blocks shown in Figure 1. Racks of peripheral equipment modules containing line cards provide access to telephone units, or to data adapter modules which provide data communication services. Trunk cards provide high-speed links to host computers, or other PBXs. Network switching modules interconnect calling parties to answering parties via digital multiplexed links. Control modules containing a Central Processor Unit (CPU), mass storage and local memory, perform the "setup" and "tear-down" of each circuit connection, as well as the monitoring of PBX activities. Central resources are modules of hardware and software which operate in a time-shared manner. The central resources include facilities such as ringing tone generator and message recording modules.

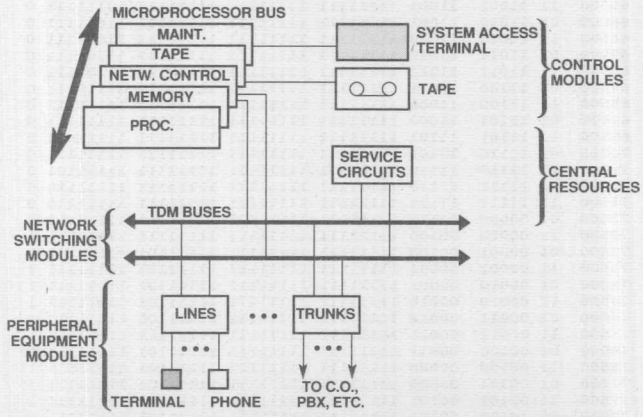


Figure 1. PBX Communication and Control Architecture

Hierarchy of Channel Multiplexing in a PBX

The main function of switching modules is to interconnect circuits between two or more parties such as in a conference call. A digital telephone unit, or a line card, contains a Coder/Decoder (CODEC) device. The CODEC device digitizes voice and transmits it into a Pulse Code Modulated (PCM) data stream on a common serial data highway at appropriate time intervals, which are assigned at the start of the call.

Analog voice is sampled at 8 KHz and passes through an eight-bit analog-to-digital converter. The digitized sample (eight-bit binary word) is transmitted serially to produce a serial data stream of (8 x 8 =) 64 Kbps. Thirty-two of these samples or CODECs are normally connected to a common highway operating at (32 x 64 =) 2.048 Mbps. Each CODEC is assigned "Serial Time Slots" of eight-bit duration, and sends its eight bits of data at the rate of 2.048 Mbps every 8 KHz. Another time slot is available on a separate highway for the receive data stream.

Therefore, a 2.048 Mbps highway can support thirty-two channels or voice connections in one direction. However, there are several ways to combine serial highways for additional voice connections. One way is to combine four highways into one high-speed highway of 8.192 Mbps. Further integration is possible, but the aggregate data rate becomes high and leads to implementation problems including timing, signal reflection, and radiation, to name a few.

Another way to integrate channels without increasing the clock rate is to convert eight serial data highways into a single eight-bit parallel bus. With this method, the serial clock rate and the parallel clock rate are the same, 2.048 Mbps. This means that data throughput on the eight serial highways of 16.384 Mbps is maintained in the eight-bit parallel bus operating at 2.048 Mbps. This works by careful alignment of the incoming parallel time slots to serial time slots on the serial highways. An eight-bit format converter scheme, sometimes called a "Corner Bender", is widely used in PBX or CO switching modules. Four of these parallel buses are combined (see Figure 2) to form a single parallel system bus that is thirty-two bits wide, yet still operates at 2.048 Mbps. The multiplexing architecture shown in Figure 2 is used in the Harris 20-20™ Integrated Network Switch (see Reference 1).

Space switching and/or time slot switching to interconnect different parties is performed either in the serial highway format or in the parallel data stream format.

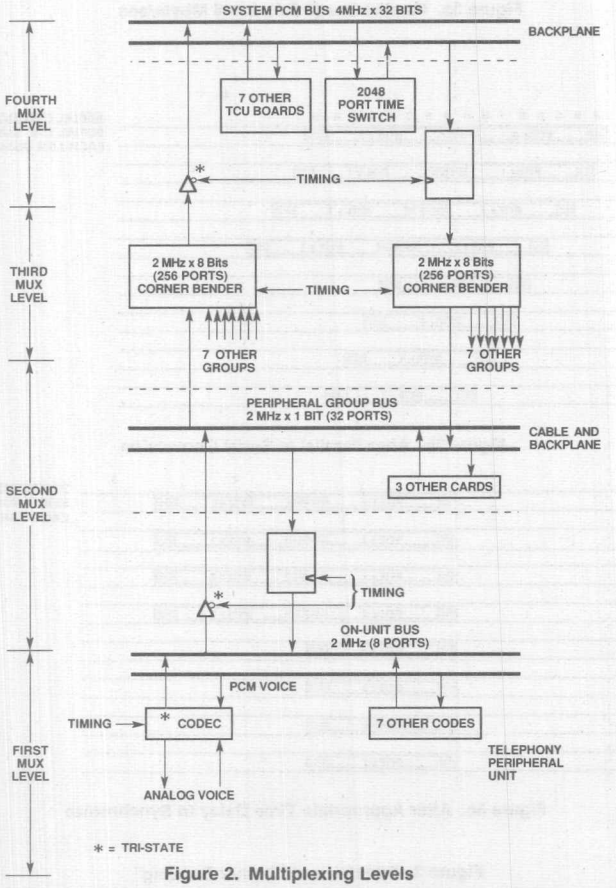


Figure 2. Multiplexing Levels

The circuit consists of eight parallel-to-serial shift registers, and additional shift registers which provide proportional delays to synchronize the outputs. The principle of operation is shown in Figure 3, and the circuit is shown in Figure 4. Parallel data from the parallel bus is loaded with each clock into one of the eight shift registers in a rotating pattern. The data then shifts out of the shift register with each subsequent clock. After all eight registers have been loaded, the first register will be empty on the next clock. New data is loaded into this register at the same clock edge as the last bit shifts out to the next stage. Thus, a continuous flow of data through the registers is insured. The data out of the eight shift registers is skewed in relation to each other as shown in Figure 3b. The various space slots on the highways need to be aligned to keep synchronization. This is achieved by proportionally delaying the outputs. Channel 1 output is delayed by seven clocks (using a seven-bit serial shift register), channel 2 is delayed by six clocks, etc. Channel 8 output does not go through any additional delay. The final outputs of the delay registers are aligned and data is clocked out in a synchronous fashion. See Figure 3c.

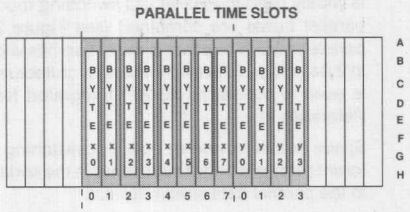


Figure 3a. Parallel Bus (8 Bit), 2.048 Mbyte/sec

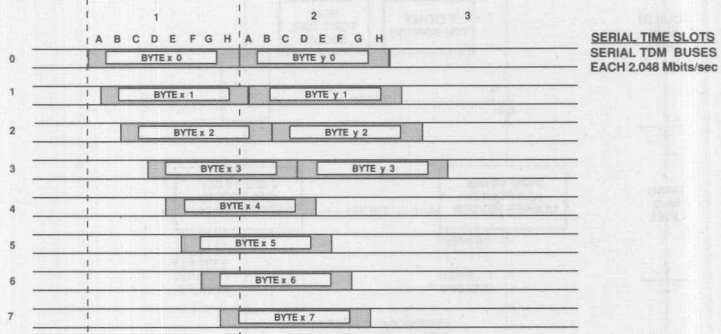


Figure 3b. After Parallel to Serial Conversion

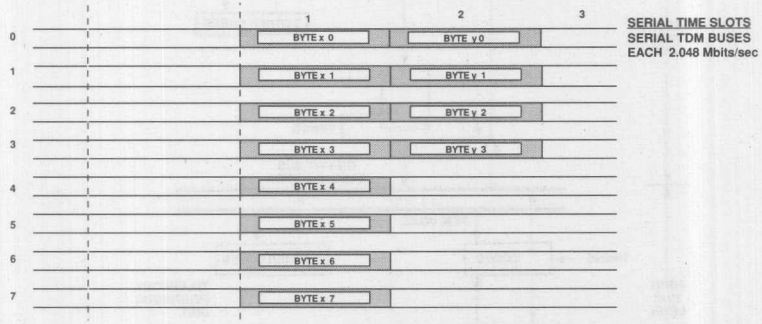


Figure 3c. After Appropriate Time Delay to Synchronize

Figure 3. Principles of "Corner Bending"

Figure 4 also shows the additional circuitry needed to control the shift registers. A preloadable three-bit counter keeps count of eight clock pulses. The parallel-to-serial bus conversion can be "programmed" to start in any register by setting the appropriate binary value on the counter preload inputs and applying a pulse to the sync input. If the loading sequence produced by the counter is not required, it can be disabled by connecting the "clock" to "sync" input. At each positive clock edge, the register loaded will depend upon the data at the counter inputs on the previous positive clock edge. The 3-to-8 decode circuit produces load pulses to latch parallel data into the shift registers. Figure 5 shows the parallel-to-serial conversion data matrix.

The description above shows the conversion of parallel data into eight streams of serial data. However, the same circuit also performs serial-to-parallel conversion. A serial eight-bit data stream on one of the eight inputs appears as an eight-bit parallel word on the eight outputs. Successive parallel words appearing at the eight outputs correspond to the serial data on each of the eight inputs in rotation. See Figure 5.

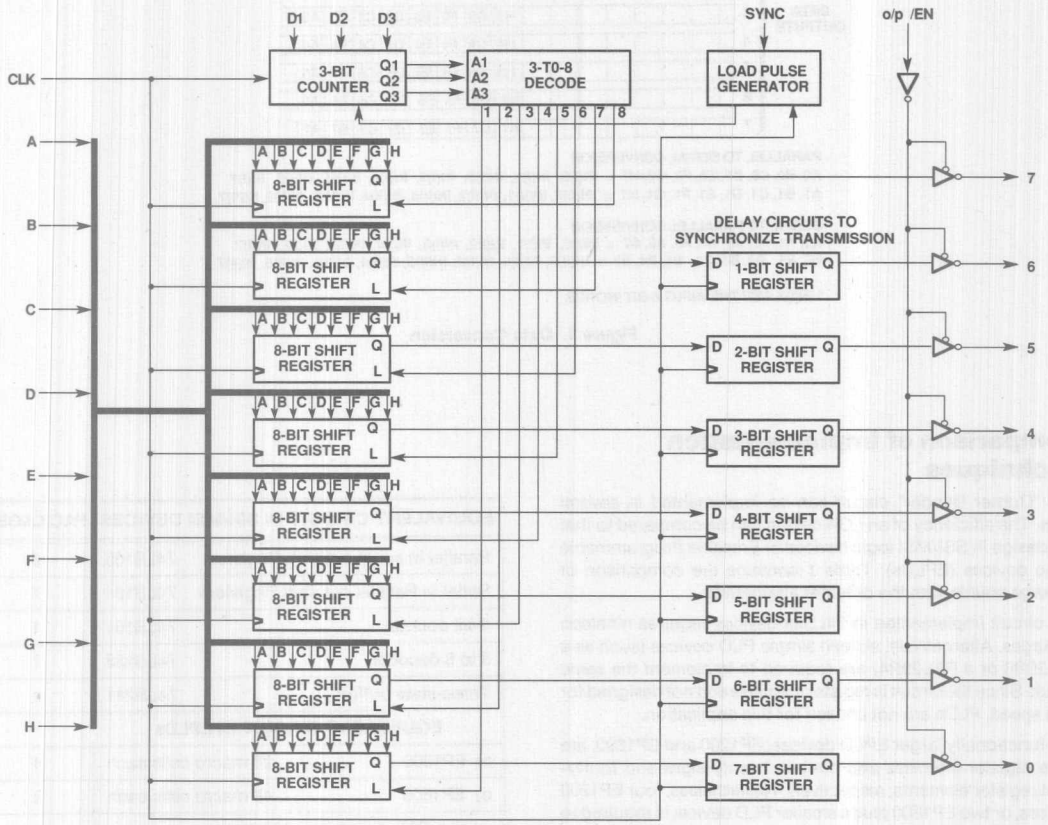
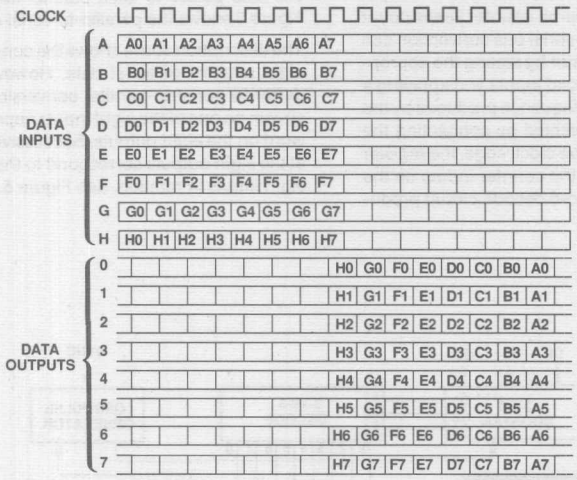


Figure 4. 8-Bit Format Converter ("Corner Bender") Circuit



PARALLEL TO SERIAL CONVERSION
A0, B0, C0, D0, E0, F0, G0, H0 = IN(I)0, IN(I)1, IN(I)2, IN(I)3, IN(I)4, IN(I)5, IN(I)6, IN(I)7
A1, B1, C1, D1, E1, F1, G1, H1 = IN(II)0, IN(II)1, IN(II)2, IN(II)3, IN(II)4, IN(II)5, IN(II)6, IN(II)7

SERIAL TO PARALLEL CONVERSION
A0, A1, A2, A3, A4, A5, A6, A7 = IN(I)0, IN(I)1, IN(I)2, IN(I)3, IN(I)4, IN(I)5, IN(I)6, IN(I)7
B0, B1, B2, B3, B4, B5, B6, B7 = IN(II)0, IN(II)1, IN(II)2, IN(II)3, IN(II)4, IN(II)5, IN(II)6, IN(II)7

* IN(I)X ARE THE INPUT 8-BIT WORDS

Figure 5. Data Conversion

Comparison of Implementation Techniques

The "Corner Bender" circuit can be implemented in several ways. The efficiency of an LCA design can be compared to that of a design in SSI/MSI logic devices or Erasable Programmable Logic devices (EPLDs). Table 1 contains the comparison of package counts with the different alternatives.

The circuit implemented in 74LSxx devices requires nineteen packages. Alternatively, sixteen simple PLD devices (such as a PAL20R8 or a PAL20R4) are required to implement the same circuit. Since the circuit is register intensive and not designed for high speed, PLDs are not chosen for this application.

The functionally larger EPLD devices, EP1200 and EP1800, are more register intensive and contain twenty-eight and fourty-eight register elements, respectively. Nevertheless, four EP1200 devices, or two EP1800 plus a smaller PLD device, is required to implement this design.

The entire Corner Bender circuit fits neatly into a single LCA 2018 device. It uses ninety-nine out of the one hundred available internal macrocells and demonstrates efficient implementation of shift registers and small counters. An LSI device from Plessey (Reference 2) is available to implement the same circuit. However, it is an NMOS device and limited to speeds of under 2 MHz.

EQUIVALENT CIRCUIT IN SSI/MSI DEVICES		PACKAGES
Parallel in serial out shift Registers	74LS165	8
Serial in Parallel out shift Registers	74LS164	7
4-bit counter	74LS161	1
3 to 8 decoder	74LS138	1
Three-state buffers	74LS241	1
EQUIVALENT CIRCUIT IN EPLDs		
a) EP1200	28 macro cells each	4
b) EP1800	48 macro cells each	2
plus PAL device e.g., 16R4		1
EQUIVALENT CIRCUIT IN MMI LCA DEVICE		
LCA 2018 device	100 macro cell	1

Table 1. Implementation Alternatives for the 8-Bit Format Converter Circuit "Corner Bender"

"Corner Bender" Design in an LCA Device

A count of the register elements in the circuit shows a need of sixty-four elements for the eight-shift registers, twenty-eight elements for the delay registers, three elements for the counter, and four elements for the 3-to-8 decode circuit. Each configurable logic block (CLB) in the LCA device can produce two outputs. Hence, only four CLBs are required for the 3-to-8 decode circuit. Since, each CLB contains one register element, the total count of CLBs is therefore ninety-nine. The LCA 2018 contains one hundred CLBs. Twenty-two input/output blocks (IOBs) were used. Figure 6 shows the layout of the design in the LCA device. The circuit fits into a 68-pin PLCC package.

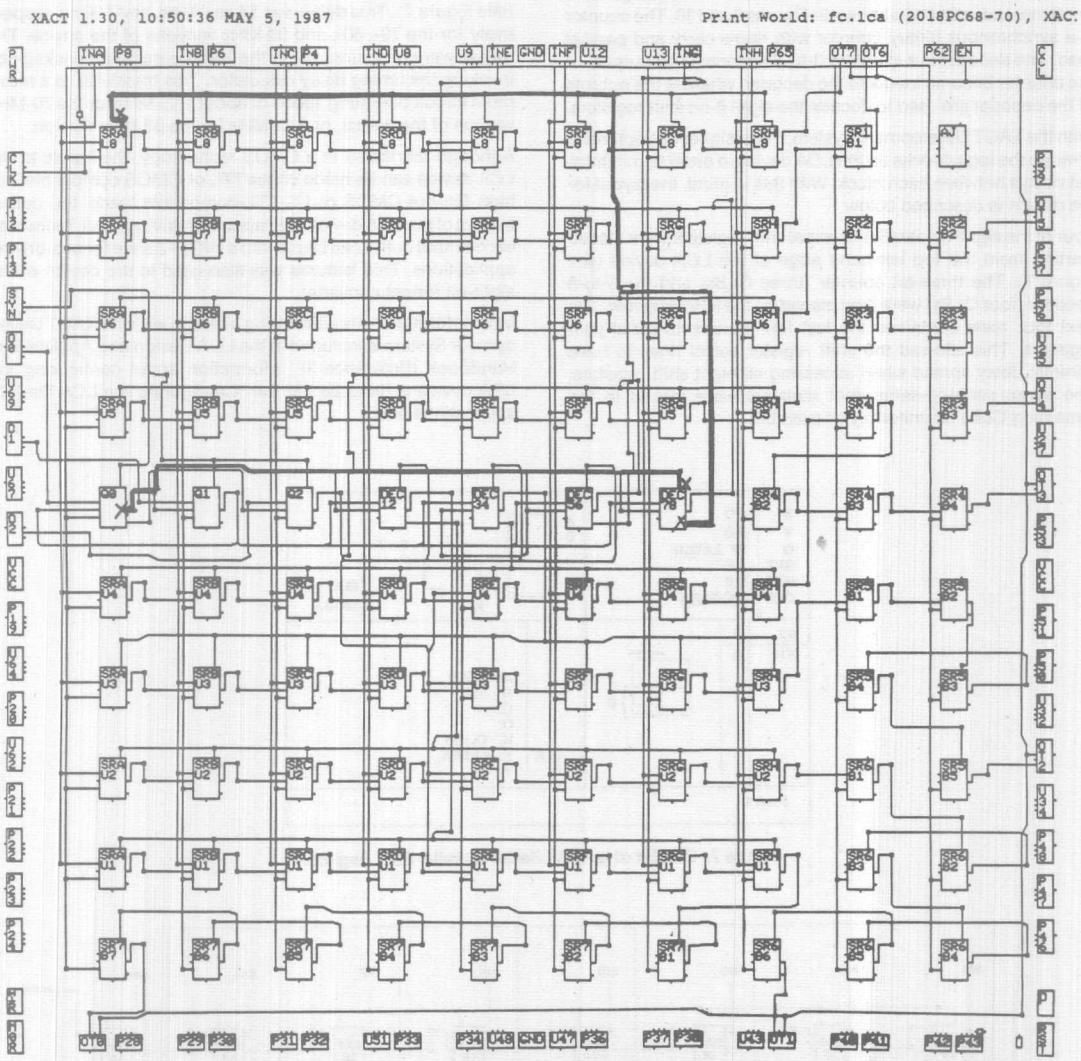


Figure 6. Layout Diagram of "Corner Bender" Circuit on LCA Device of Size 10x10

eight-bit shift register in this design is shown in Figure 7. Input A is the decoder input and it is ANDed with input B, the data to be shifted. To implement the eight-bit shift registers from the one-bit shift register, eight of the one-bit shift registers were linked together so that the output of each register became an input to the next register. This is shown in Figure 8 where each CLB represents one bit of the eight-bit shift register.

The three-bit counter and the 3-to-8 decoder in this design were implemented in CLBs as shown in Figures 9 and 10. The counter is a synchronous binary counter with ripple carry and parallel load. The decoder is a standard 3-to-8 decoder. The outputs of the counter become inputs to the decoder, whereas the outputs of the decoder are used to decode the eight 8-bit shift registers.

With the XACT Development System, the designer can optimally arrange the logic blocks on the LCA device in order to minimize net delays between each block. With this in mind, the layout for the design is described below:

Four of the eight-bit parallel-to-serial shift registers were placed starting from the top left-hand edge of the LCA device (see Figure 7). The three-bit counter (three CLBs) and the 3-to-8 decoder (four CLBs) were then placed on the following row. The next four rows contained the last four parallel-to-serial shift registers. This allowed the shift register select lines to have minimal delay spread when accessing all eight shift registers. The seven serial-to-serial shift registers were placed in the remaining CLBs as uniformly as possible.

System. Blocks placed in close proximity can use local interconnection resources which incur short signal propagation delays, whereas blocks placed further apart must use either "long lines" or other interconnection resources. Manual optimization using the delay efficient "long lines" was performed for the most critical net connections. After routing completion, the longest delay between two clock pulses was the delay for the counter to change state, the state which is decoded via the 3-to-8 decoder and selects the appropriate shift register to load the parallel data (see Figure 7). This delay was 54 ns, 79 ns, and 106 ns, respectively for the 70-, 50-, and 33-MHz versions of the device. The delays were measured using the XACT simulation package by invoking the timing delay calculator. This translates to a maximum circuit operating speed of about 18.5 MHz for the 70-MHz version of the device, or 9.43 MHz for the 33 MHz version.

Although fabricated in a CMOS technology, the inputs to the LCA device can be made either TTL or CMOS compatible. For high fan-out CMOS or LS TTL-compatible loads, the output buffers of the LCA device are capable of driving 4 mA. Moreover, each output buffer can be put into a HIGH-Z state for bus-driving applications. This features was also used in the design of the eight-bit format converter.

More information about entering a design with the XACT Development System is included in the LCA Design and Applications Handbook (Reference 3). Information about configuring the LCA device is described in the "Configuring the LCA Device" applications note.

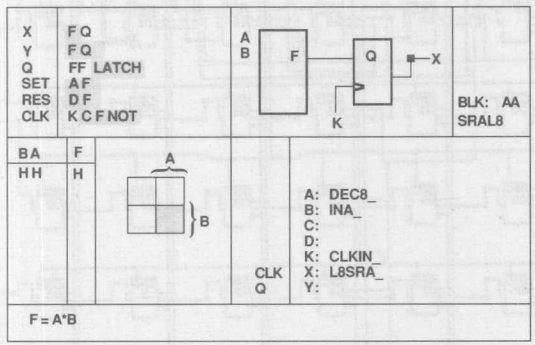


Figure 7. One Bit of an 8-Bit Serial - Parallel Shift Register

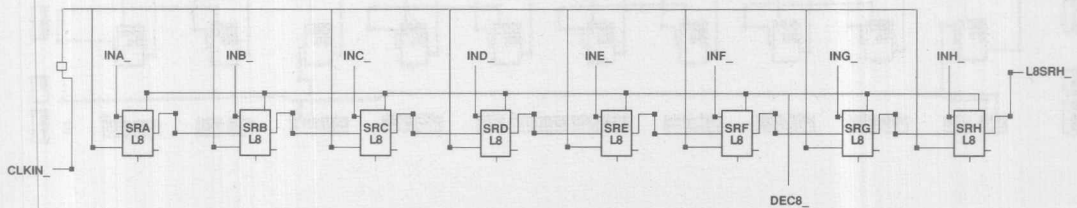


Figure 8. CLB Schematic Output for the 8-Bit Shift Register

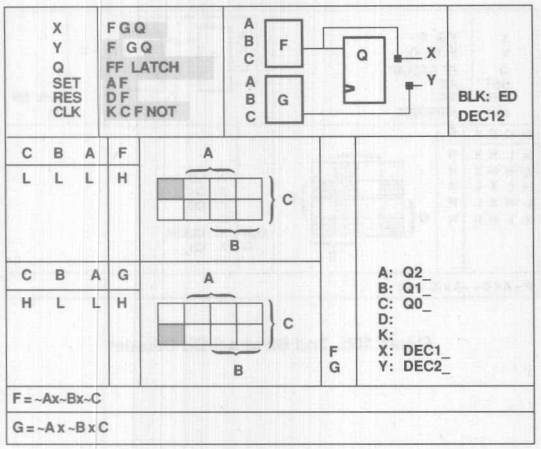


Figure 9a. Decode Outputs 1 and 2 of 3-to-8 Decoder

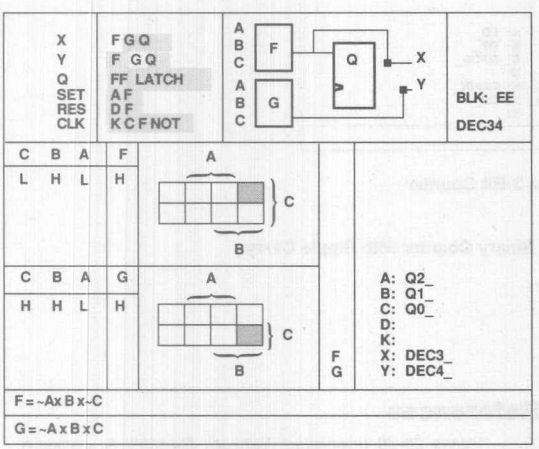


Figure 9b. Decode Outputs 3 and 4 of 3-to-6 Decoder

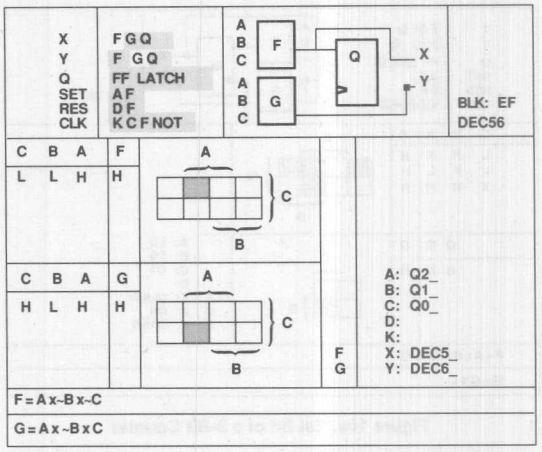


Figure 9c. Decode Outputs 5 and 6 of 3-to-8 Decoder

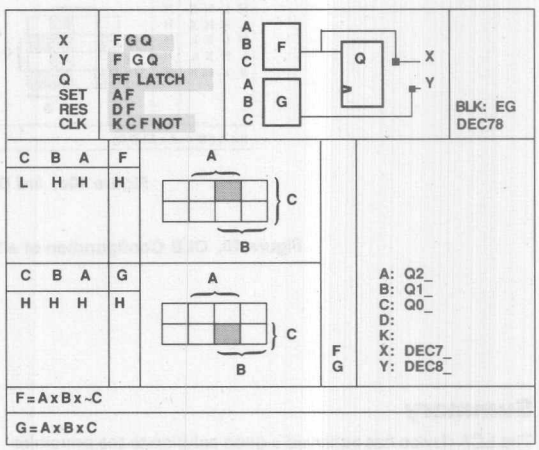


Figure 9d. Decode Outputs 7 and 8 of 3-to-8 Decoder

Figure 9. CLB Configuration of a 3-to-8 Decoder

2

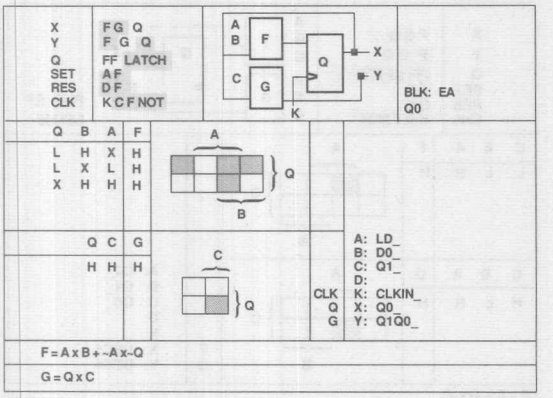


Figure 10a. 1st Bit of a 3-Bit Counter

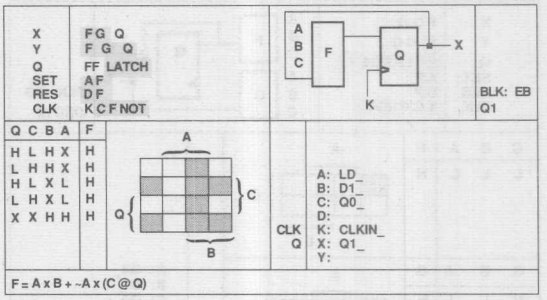


Figure 10b. 2nd Bit of a 3-Bit Counter

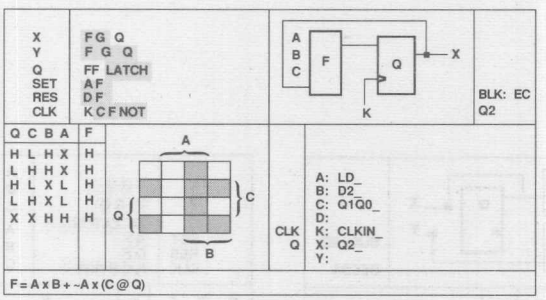


Figure 10c. 3rd Bit of a 3-Bit Counter

Figure 10. CLB Configuration of a 3-Bit Binary Counter with Ripple Carry

Summary

The LCA device has achieved a good solution to the principles used for multiplexing digitally-encoded voice channels in both serial and parallel hierarchies. These hierarchies can accommodate thousands of voice circuits in a PBX switching module. The detailed design of the eight-bit (parallel-to-serial and serial-to-parallel) format converter circuit, or Corner Bender, developed in Monolithic Memories' LCA M2018 device was shown to be efficiently implemented. Despite the fast development of the system using the XACT Design Editor, the final design was programmed and bench tested to verify functional integrity.

The Corner Bender design is available from Monolithic Memories upon request. The bit pattern and .LCA file will be provided for programming the LCA device in an EPROM. Please ask for design XDES05.LCA

References

1) *"Harris 20-20 Integrated Network Switch"*, A. Jackson, IEEE SAC-3, No. 4 July 1985, p561-568.

2) MJ1410, 8-bit Format Converter, Data Sheet, Plessey Semiconductors.

3) Logic Cell Array Design and Applications Handbook, Monolithic Memories, 1987.

Serial Data Link Controller

AN-131E

Functional Description

This application was first developed by LTT, Conflans Ste. Honorine, France. Part of the design, reprinted with courtesy of LTT, is used to control a serial data link based upon a specialized LSI chip.

Originally designed with six standard SSI/MSI circuits, this same function can now be implemented, not only into a single PAL20RA10, but with even more features and better performance. The function can be divided into three sub-functions:

1. Address Decoding
2. Control Flags
3. Transmission Speed Selection

Up to four address lines are allowed (eight were actually used), plus two extra lines which are special decoding controls (MEM/IO selection, Enable Control . . .). Two flip-flops load flag conditions, from the address bus (A1 and A2), providing handshake between the 6850 UART and the communication lines. They have a common clock which also serves as Chip Select (CSO) for the UART.

The UART Transmit clock (TXCLK) can be directly connected to the Receive Clock (CK or RXCLK) or represents the Receive Clock value divided by sixteen. This function was performed by four "D" Flip-Flops connected as a 4-stage Asynchronous Divider. Since each basis cell in a PAL20RA10 has four Product Terms available, this function could be implemented either asynchronously or synchronously. In the PAL Design Specification example, a 4-bit synchronous divider was used instead of the asynchronous circuit shown in the schematic.

Pin Description

1. TEST ..... Allows preload function for testing.
2. SYSRESET ..... Reset line from microprocessor.
3. A2 ..... Address line from address bus.
4. A1 ..... Address line from address bus.
5. HDSHAKE ..... Handshake line (CTS/RTS).
6. CK ..... External clock.
7. E ..... Enable line from microprocessor.
8. AUXDECOD ..... Extra decoding line (e.g. board level decoding).
9. A3 ..... Address line from address bus.
10. A3 ..... Address line from address bus.
11. A3 ..... Address line from address bus.
12. GND ..... Reference power supply ground.
13. /OE ..... Output enable line.
14. A6 ..... Address line from address bus.
15. SPEEDSEL ..... Speed selection line.
16. DIV4 ..... MSB 4-bit synchronous counter.
17. DIV3 ..... 3rd stage synchronous counter.
18. DIV2 ..... 2nd stage synchronous counter.
19. DIV1 ..... LSB 4-bit synchronous counter.
20. SC0 ..... UART chip select line (CSO).
21. BLOCREC ..... Bloc receive line.
22. DIR DIV ..... Direct or divided clock.
23. /TPH ..... External use flag.
24. VCC ..... 5 V power supply.

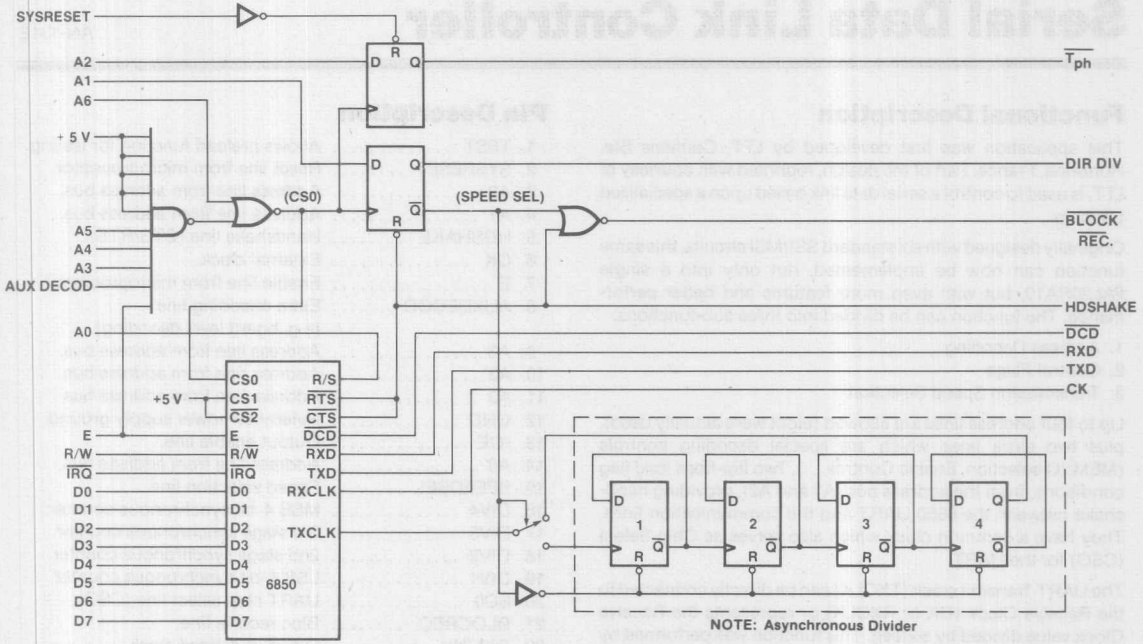


Figure 1.

After

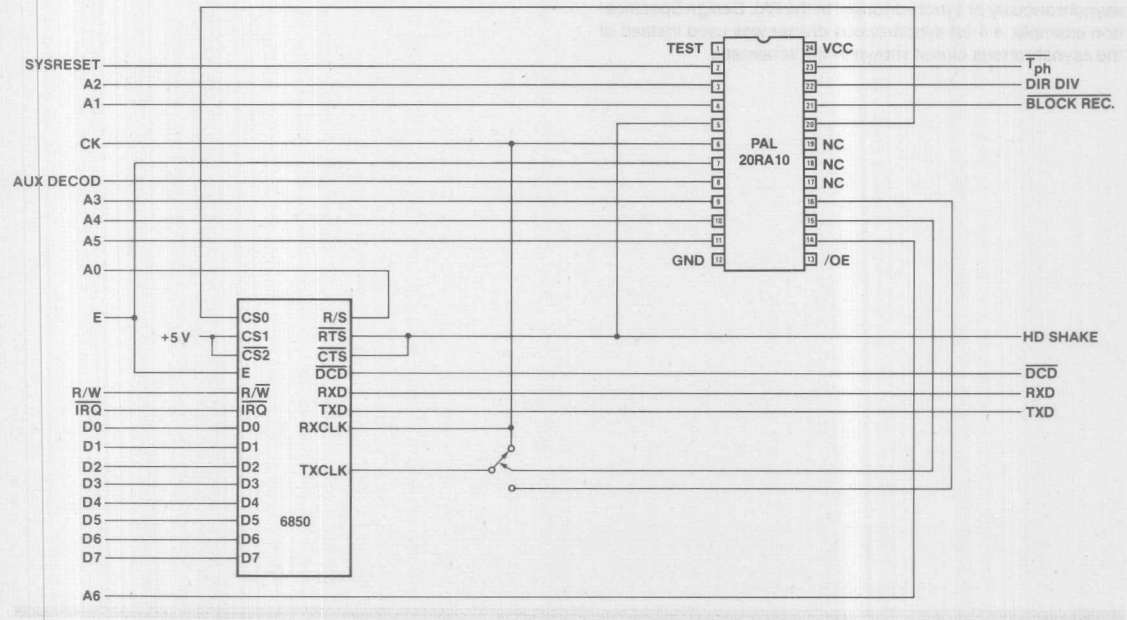


Figure 2.

```
Title     Serial Data Link Controller
Pattern   Link.pds
Revision  A
Author    Jose Juntas / Kelvin Chow
Company   Monolithic Memories Inc., Santa Clara, Ca
Date      3/1/85

CHIP SE_CH_CNTRL  PAL20RA10

TEST SYSRESET A2 A1 HDSHAKE CK E AUXDECOD A3 A4 A5 GND
/OE A6 SPEEDSEL DIV4 DIV3 DIV2 DIV1 CSO BLOCREC DIRDIV
/TPH VCC

EQUATIONS

/TPH              := A2                      ;Load A2 as flag
/TPH.CLKF          = CSO                     ;CLK W/ ADDR. decode
/TPH.SETF          = SYSRESET                ;global system reset

DIRDIV            := A1                      ;Load speed ratio
DIRDIV.CLKF        = CSO                     ;CLK W/ ADDR. decode
DIRDIV.SETF        = /HDSHAKE                ;CLR by CTS/RTS line

/BLOCREC           = /DIRDIV                 ;Controlled by speed
                   + HDSHAKE                 ;option and CTS/RTS
                                             ;line
CSO                = /A6*A5*A4*A3*AUXDECOD*E
                                             ;UART address valid
/DIV1             := DIV1                    ;4-bit synchronous
                                             ;divider LSB
/DIV1.CLKF         = CK                      ;CLK by CK(external)
/DIV1.SETF         = /DIRDIV                 ;CLR by speed option

/DIV2             := /DIV1*/DIV2             ;2ND stage of
                   + DIV1*DIV2               ;divider
/DIV2.CLKF         = CK                      ;CLK by CK(external)
/DIV2.SETF         = /DIRDIV                 ;CLR by speed option

/DIV3             := /DIV2*/DIV3             ;3RD stage of
                   + /DIV1*/DIV3             ;divider
                   + DIV1*DIV2*DIV3
/DIV3.CLKF         = CK                      ;CLK by CK(external)
/DIV3.SETF         = /DIRDIV                 ;CLR by speed option

/DIV4             := /DIV3*/DIV4             ;4TH stage of
                   + /DIV2*/DIV4             ;divider MSB
                   + /DIV1*/DIV4
                   + DIV1*DIV2*DIV3*DIV4
/DIV4.CLKF         = CK                      ;CLK by CK(external)
/DIV4.SETF         = /DIRDIV                 ;CLR by speed option

SPEEDSEL          := /A1                     ;Load speed choice
SPEEDSEL.CLKF      = CSO                     ;CLK W/ ADDR. decode
SPEEDSEL.SETF      = /HDSHAKE                ;CLR by CTS/RTS line

SIMULATION

TRACE_ON A1,A2,A3,A4,A5,A6,E,              ;Signals to be
         AUXDECOD,SYSRESET,/TPH,HDSHAKE,          ;observed
         CSO,SPEEDSEL,DIRDIV,CK,
         DIV1,DIV2,DIV3,DIV4
SETF SYSRESET,/HDSHAKE                     ;Reset all regs

CHECK /SPEEDSEL,/DIRDIV,TPH
SETF /SYSRESET,A1,A2,A3,A4,A5,/A6,HDSHAKE,  ;Set decode
    E,AUXDECOD                              ;condition

CHECK /SPEEDSEL,DIRDIV                     ;Check SPEEDSEL and
                                           ;DIRDIV regs

FOR I:=1 TO 15 DO
 BEGIN                                     ;This portion
  SETF CK                                  ;simulates divide
                                           ;by four counter
  SETF /CK
 END
```

Simulation Results

```
Page :  1
                g g  g gg  gg gg gg g g gg gg gg  gg gg gg
 A1         XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 A2         XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 A3         XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 A4         XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 A5         XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 A6         XXLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 E          XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 AUXDECOD   XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 SYSRESET   HHLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
/TPH        LLLLHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 HDSHAKE    LLHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 CSO        XXHHHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 SPEEDSEL   LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL LLLLLLLLLL
 DIRDIV     LLLLHHHHHH HHHHHHHHHH HHHHHHHHHH HHHHHHHHHH
 CK         XXXXXHHLHH LHHLHHLHHL HHLHHLHHLH HLHHLHHLHH
 DIV1       XLLLLLHHHL LLHHHLLLHH HLLLHHHLLL HHHLLLHHHL
 DIV2       XLLLLLLLLH HHHHHLLLLL LHHHHHHLLL LLLHHHHHHL
 DIV3       XLLLLLLLLL LLLLLHHHHH HHHHHHHLLL LLLLLLLLLH
 DIV4       XLLLLLLLLL LLLLLLLLLL LLLLLLLHHH HHHHHHHHHH

Page :  2
                gg gg gg g
 A1         HHHHHHHHHH
 A2         HHHHHHHHHH
 A3         HHHHHHHHHH
 A4         HHHHHHHHHH
 A5         HHHHHHHHHH
 A6         LLLLLLLLLL
 E          HHHHHHHHHH
 AUXDECOD   HHHHHHHHHH
 SYSRESET   LLLLLLLLLL
/TPH        HHHHHHHHHH
 HDSHAKE    HHHHHHHHHH
 CSO        HHHHHHHHHH
 SPEEDSEL   LLLLLLLLLL
 DIRDIV     HHHHHHHHHH
 CK         LHHLHHLHHL
 DIV1       LLHHHLLLHH
 DIV2       LLLLLHHHHH
 DIV3       HHHHHHHHHH
 DIV4       HHHHHHHHHH
```

QAM Encoder in a ZPAL Device

AN-166

The ZPAL PALC20R8Z family consists of four devices: combinatorial PALC20L8Z, and registered PALC20R8Z, PALC20R6Z, and PALC20R4Z. These devices offer zero standby and very low active power dissipation, along with traditional architecture and functionality of standard PAL devices. These 24-pin devices offer high speed (TPD = 35 ns) and are ideal for applications in communications equipment and instrumentation, which are typically under severe power constraints.

This note describes one such application in communications equipment for Quadrature Amplitude Modulation (QAM) encoding and provides a design based on a PALC20R8Z device. QAM encoding technique is used in standalone modem equipment which uses CMOS devices. Monolithic Memories' CMOS ZPAL devices became a natural choice for replacing glue logic or any complete function that is being implemented in discrete SSI components.

QAM Encoder

Data modems use encoders as a means of increasing data throughput within a given bandwidth of a channel. Quadrature Amplitude Modulation (QAM) gives an input data rate of four times the encoding rate, which is the output rate of the encoder. The encoding rate determines the channel bandwidth required by the output signal. Voice band data modems have become popular for use over the switched telephone network for interconnecting personal computers to distant computers. In these modems the data encoder function is usually integrated within the modem chip set. However, higher-speed modems for 20-MHz digital radio, satellite up/down links and other group band modems implement the encoder as a discrete hardware function due to the speed requirements. A PALC20R8Z device is ideally suited for implementing the data encoder function because of its ability to implement state machines efficiently and its low power consumption.

The PAL Design Specification (QAM.PDS) file enclosed shows the QAM Encoder example implemented using the PALASM®2 software state machine syntax.

Modem Encoding Techniques:

Figure 1 shows the encoder function within a typical modem transmitter. Incoming data may be scrambled (to randomize the data) and then passed through the encoder. The output streams (I, Q) are passed through pulse-shaping filters before being multiplied by the sine and cosine carrier frequencies.

Let us consider the QAM encoding technique as used in the voice band data modems. The scheme is also widely used in other high-speed applications. Some of the popular modem standards for the switched telephone network are Bell 212A (for 1200-bps operation), CCITT V.22 (for 1200-bps operation), CCITT V.22bis (for 2400-bps operation) and CCITT V.32 (for 9600-bps operation).

According to Bell 212A, CCITT V.22 and V.22bis recommendations, full duplex communication over a pair of telephone wires is achieved by separating the bandwidth occupied by the signals traveling in opposite directions. (see Figure 2).

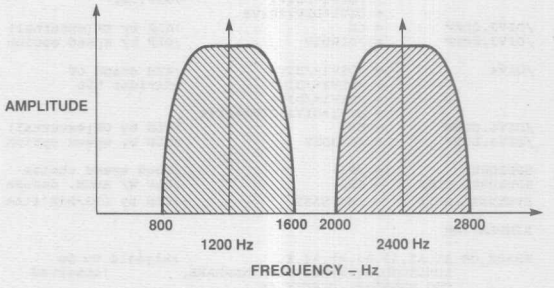


Figure 2. Full Duplex Communication by Bandwidth Separation

The transmit carrier frequency of one modem is 1200 Hz while the far-end modem uses 2400 Hz as its transmit carrier frequency. The encoding rate (or symbol rate) for each carrier is 600 baud (equivalent to transitions per second) and the band-

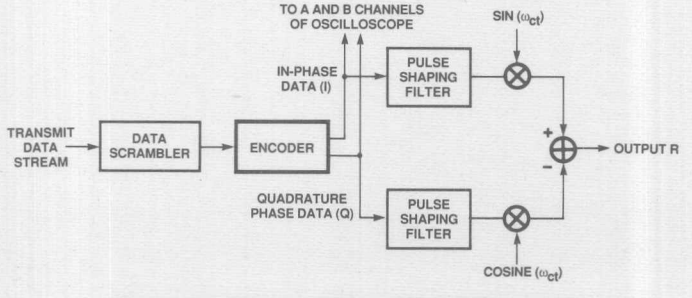


Figure 1. Data Encoder in a Typical Modem Transmitter

width occupied by the outgoing signal spectrum is about 800 Hz, centered around the carrier. Since the two bands of transmission do not overlap, no interference takes place between the two channels of communications.

The transmit data throughput in each direction is increased from the 600 baud encoding rate by means of encoding two or four incoming data bits at a time. This produces a multilevel signal which is then used to modulate the carriers. The transmit data bit rate is then 1200 bits per second (bps) and 2400 bps respectively.

CCITT V.22bis and V.32 recommendation modems employ Quadrature Amplitude Modulation (QAM) technique for modulating the carrier frequency. The 16-state QAM signal constellation can be observed at the I and Q outputs (Figure 1) of the encoder. The diagram of Figure 3 is observed by connecting the I and Q outputs to the A and B channels of a dual channel oscilloscope and turning the timebase to the X-Y setting.

The data stream to be transmitted is divided into groups of four consecutive bits called quadbits. The first two bits are encoded as a phase quadrant change relative to the quadrant occupied by the preceding signal element. See Figure 3 and Table 1. The last two bits of each quadbit define the amplitude of the signal element in each phase quadrant. The left-hand bits in Table 1 are the first of each pair in the data stream as it enters the encoder after the data scrambler.

Bell 212A and CCITT V.22 recommendation modems employ the Differential Quadrature Phase Shift Keying (DQPSK) technique for modulating the carrier. The data stream to be transmitted is divided into groups of two consecutive bits called dibits. The dibits are encoded as a phase quadrant change relative to the quadrant occupied by the preceding signal element. The signal element corresponding to XX01 (Y0, Y1, X2, X3) in the V.22bis signal constellation (Figure 3) is transmitted irrespective of the quadrant occupied. This ensures compatibility with the V.22bis modems operating in a fallback mode of 1200 bps.

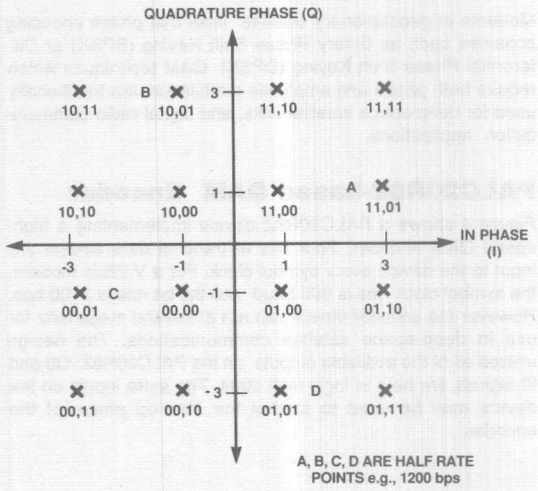


Figure 3. QAM Signal Pattern (Y0, Y1, X2, X3) e.g., 2400 bps (V.22bis)

FIRST TWO BITS OF QUADBITS OR A DIBIT (X0, X1)	PHASE QUADRANT CHANGE
00	+ 90
01	0
11	270
10	180

Table 1. Relative Quadrant Change for Input (X0, X1)

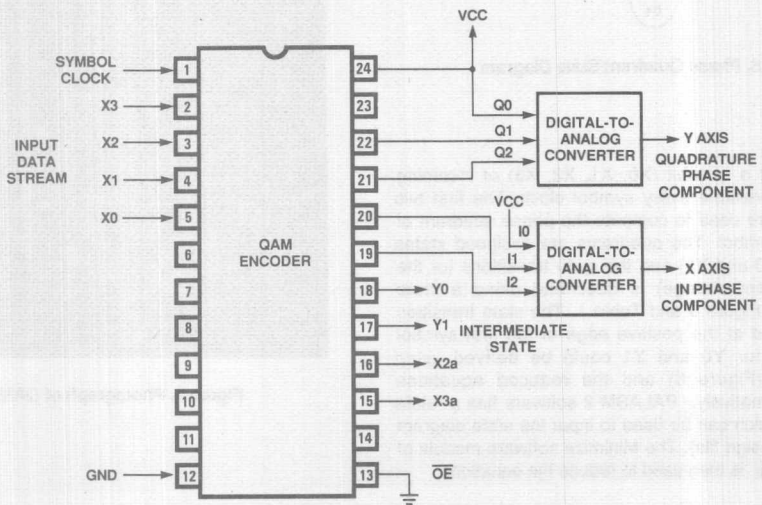


Figure 4. PALC20R8Z Device QAM Encoder

ferential Phase Shift Keying (DPSK). QAM techniques which require both phase and amplitude modulation, are traditionally used for deep-space satellite links, and digital radio communication applications.

PALC20R8Z-based QAM Encoder

Figure 4 shows a PALC20R8Z device implementing a high-speed QAM encoder. Four bits of transmit data stream are input to the device every symbol clock. For a V.22bis modem, the symbol clock rate is 600 baud and the bit rate is 2400 bps. However the encoder device can run at several megahertz for use in deep-space satellite communications. The design utilizes all of the available outputs on the PALC20R8Z. Q0 and I0 signals are held in logic high state. The extra inputs on the device may be used to control the start-up phase of the encoder.

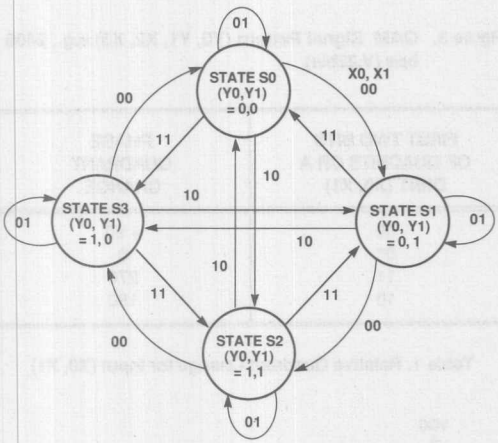
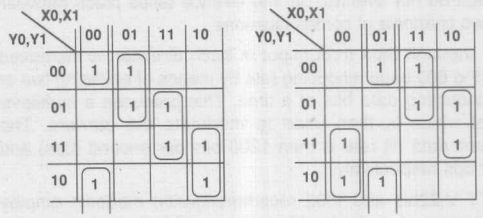


Figure 5. Phase Quadrant State Diagram

It is assumed that a quadbit (X0, X1, X2, X3) of incoming transmit data is available every symbol clock. The first two bits, X0 and X1, are used to compute the phase quadrant of the next output symbol. The quadrants are assigned states (represented by Y0 and Y1) and the state transitions (or the differential quadrant change) is described using a state diagram shown in Figure 5 and Table 1. The state transition outputs are latched at the positive edge of the first symbol clock. Equations for Y0 and Y1 could be derived using Karnaugh maps (Figure 6) and the reduced equations implemented. Alternatively, PALASM 2 software has a state machine syntax which can be used to input the state diagram directly (see the design file). The Minimize software module of PALASM 2 software is then used to reduce the equations.



Karnaugh Map for Next /Y0 State

Karnaugh Map for Next /Y1 State

Figure 6. Karnaugh Maps for Next States of /Y0, /Y1

The last two bits, X2 and X3, are latched at the same time as the new states are computed (see design file). The output data values, I and Q, are combinatorial functions of Y0, Y1, X2, and X3. These outputs are latched at the positive edge of the next symbol clock. Thus the computational delay through the encoder is two symbol clocks. If the processing delay is critical to the design, it could be reduced to one symbol clock by using a PAL20RA10 device because this device allows independent clocking of registers. The phase quadrant state could then be latched on the positive edge of the symbol clock and the output could be available on the negative edge of the clock.

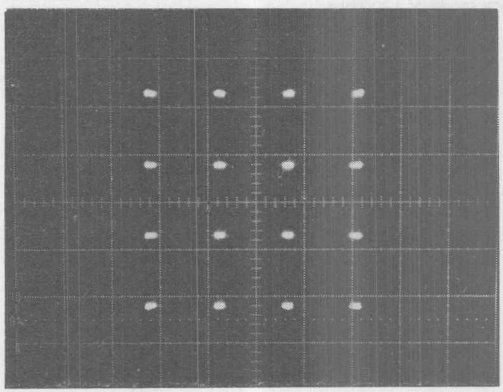


Figure 7. Photograph of QAM Pattern

The values +3, +1, -1, -3 of I and Q (Figure 3) are represented by the three-bit binary words, 111, 101, 011, 001, (I2 I1 I0 or Q2 Q1 Q0), respectively. These three-bit values are output every symbol. This number scheme allows the use of simple digital-to-analog conversion as required in the satellite applications. For these high-speed applications the pulse-shaping filtering and carrier-modulation functions are performed in analog domain using operational amplifiers and active components.

The PALC20R8Z device has only eight registered outputs, whereas this design needs ten registered outputs (two for the phase quadrant state, and eight for the I and Q outputs). Fortunately the I0 and Q0 are always "1" (as defined above) and hence can be tied to this state at the input to the next stage. In this case the eight registers in the PALC20R8Z device are sufficient to implement the full encoder function.

The design file shows an example of using the state machine syntax of PALASM 2 software. The QAM encoder is a standard Mealy type of state machine. However, to keep the number of product terms per output reasonable (less than eight for the PALC20R8Z device), it is implemented as a Moore state machine followed by the output decode function (see appendix). There is a pipeline delay between the new state values and their corresponding outputs. The differential quadrant change is implemented in the State section of the QAM.PDS file whereas output decoding is done in the Equations section of the same file.

In the laboratory verification, the I (I0 to I2) and Q (Q0 to Q2) outputs were connected to two digital-to-analog converters. The analog outputs were then connected to the A and B channels of an oscilloscope and the X-Y pattern was observed (see Figure 7). For high data-rate operation (several MHz), the PALC20R8Z device encoder together with a data scrambler implemented in hardware facilitates a cost-effective solution for a modem transmitter.

```
TITLE           QAM_ENCODER
PATTERN         QAM.PDS
REVISION        A
AUTHOR          RAJ PARIPATYADAR, M. GZOWSKI
COMPANY         MONOLITHIC MEMORIES INC., SANTA CLARA
DATE            5 April 1987

CHIP    QAM PAL20R8
        CK X3 X2 X1 X0 NC NC NC NC NC NC GND
        /OE NC X3A X2A Y1 Y0 I2 I1 Q2 Q1 NC VCC

STATE

MOORE_MACHINE

STATE_0 := CS1 ->STATE_1 + CS2 ->STATE_0 + CS3 ->STATE_3 +->STATE_2
STATE_1 := CS1 ->STATE_2 + CS2 ->STATE_1 + CS3 ->STATE_0 +->STATE_3
STATE_2 := CS1 ->STATE_3 + CS2 ->STATE_2 + CS3 ->STATE_1 +->STATE_0
STATE_3 := CS1 ->STATE_0 + CS2 ->STATE_3 + CS3 ->STATE_2 +->STATE_1

;  Equations for the next states represented by Y0 Y1
;  These outputs are latched on the positive edge of the
;  first symbol clock

     STATE_0 = /Y0 * /Y1
     STATE_1 = /Y0 *  Y1
     STATE_2 =  Y0 *  Y1
     STATE_3 =  Y0 * /Y1


CONDITIONS
     CS1 = /X0 * /X1
     CS2 = /X0 *  X1
     CS3 =  X0 *  X1
     CS4 =  X0 * /X1


EQUATIONS


; latch values of X2 and X3 on the same symbol clock

/X3A := /X3

/X2A := /X2


;  The following  four outputs are  latched on the positive edge
;  of the next symbol clock.  Therefore the ouput of the whole
;  encoder is available after the second symbol clock.
;  The value of I0 and Q0 are HIGH and therefore input to the next
;  stage may be tied HIGH.

/I2 := STATE_0 + STATE_1
```

```
/I1 := STATE_0* X3A
     + STATE_1* X2A
     + STATE_2*/X3A
     + STATE_3*/X2A

/Q2 := STATE_0 + STATE_3


/Q1 := STATE_0* X2A
     + STATE_1*/X3A
     + STATE_2*/X2A
     + STATE_3* X3A

SIMULATION

TRACE_ON    OE  X0  X1  X2  X3  X2a  X3a  Y0  Y1  CK
            I1  I2  Q1  Q2

SETF    CK  OE                      ;Preset polarity and enable output
PRLDF   X3A X2A /Y1  /Y0 I2 I1 Q2 Q1
                                    ;check the state transitions from
SETF    /X0  /X1  /X2  /X3  /CK
                                    ;state Y0=0 Y1=0 with input
CLOCKF                              ;X0=0 and X1=0
CLOCKF                              ;X2=0 and X3=0 sets the smallest
CLOCKF                              ;vector amplitude
CLOCKF                              ;output available on the negative
                                    ;edge of clock


PRLDF   X3A X2A /Y1  /Y0 I2 I1 Q2 Q1
                                    ;check the state transitions from
SETF    X0  X1                      ;state Y0=0  Y1=0 with inputs
CLOCKF                              ;X0=1 and X1=1
CLOCKF
CLOCKF
CLOCKF

PRLDF   X3A X2A /Y1  /Y0 I2 I1 Q2 Q1
                                    ;check the state transitions from
SETF    X0  /X1                     ;state Y0=0 Y1=0 with inputs
CLOCKF                              ;X0=1 and X1=0
CLOCKF

PRLDF   X3A X2A Y1 /Y0 I2 I1 Q2 Q1
                                    ;check the state transitions from
CLOCKF                              ;state Y0=0 Y1=1 with inputs
CLOCKF                              ;X0=1 and X1=0

PRLDF   X3A X2A /Y1 /Y0 I2 I1 Q2 Q1
                                    ;check the state transition from
SETF    /X0 X1                      ;state Y0=0 Y1=0 with inputs
CLOCKF                              ;X0=0 and X1=1
```

```
                                    ;check the state transitions from
CLOCKF                              ;state Y0=0 Y1=1 with inputs
                                    ;X0=0 and X1=1

PRLDF   X3A X2A /Y1 Y0 I2 I1 Q2 Q1

                                    ;check the state transitions from
CLOCKF                              ;state Y0=1 Y1=0 with inputs
                                    ;X0=0 and X1=1

PRLDF   X3A X2A Y1 Y0 I2 I1 Q2 Q1
                                    ;check the state transitions from
CLOCKF                              ;state Y0=1 Y1=1 with inputs
                                    ;X0=0 and X1=1


PRLDF   X3A X2A /Y1 /Y0 I2 I1 Q2 Q1
                                    ;In state Y0=0 Y1=0
SETF    /X0 X1 /X2 /X3              ;check the 4 vector amplitudes
CLOCKF                              ;
CLOCKF

SETF    /X2 X3
CLOCKF
CLOCKF

SETF    X2 /X3
CLOCKF
CLOCKF

SETF    X2 X3
CLOCKF
CLOCKF

SETF    /X2 /X3
CLOCKF
CLOCKF

TRACE_OFF
```

PAL Devices Implement the Full V.32 Convolution Encoder

AN-171

The continuing drive for higher communication data rates is resulting in the development of data modems complying with CCITT specification V.32. This standard uses the Quadrature Amplitude Modulation (QAM) encoding scheme to achieve a data rate of 9600 bps while restricting the modulation rate to 2400 Hz. However, this high rate of transmission is sensitive to random errors caused by Gaussian noise on the telephone lines. Forward error correction by means of convolution encoding and Viterbi decoding improves the bit-error rate performance in the presence of white Gaussian noise. The CCITT V.32 (see Reference 1) specification provides for the optional use of a convolution encoding scheme whenever the background noise is high. A PAL20X8A and a PAL20RS8 can implement this complex encoder function in a cost-effective manner, thereby providing a simple way of implementing the transmitter of a V.32 modem. This design, using PAL devices, provides a way of implementing a separate modem transmitter synchronized only to the transmit clock. The receiver of the modem can then be implemented on a separate digital signal processing chip synchronized to the receive clock.

Implementation of the V.32 Convolution Encoder

Figure 1 shows the position of the encoder function within the simplified V.32 modem transmitter. Incoming data is scrambled to randomize the data and then passed through the encoder. The In-phase and Out-of-phase (I, Q) output data streams are passed through pulse-shaping filters before being multiplied with the sine and cosine carrier frequencies. If I and Q values are applied to the X and the Y axes of an oscilloscope with the time base turned to the X-Y setting, a signal constellation is observed. This signal pattern (photograph in Figure 8) indicates the modulation amplitude and the phase of the outgoing 2400 Hz carrier frequency. The standard signal constellation for the V.32 is the QAM scheme and is shown in Figure 2. This scheme produces sixteen states or points on the signal constellation map. The diagram may be observed by connecting the I and Q outputs of the encoder (Figure 1) to the A,B channels of an oscilloscope and turning the time base to the X-Y setting. The optional convolutional encoding scheme specified in the V.32 standard produces thirty-two points in the signal constellation (Figure 3).

The points in Figure 3 are closer to each other than the points in the straight QAM scheme depicted in Figure 2, and it may not be obvious why the second scheme should give better performance in the presence of noise. The convolution encoder, with its implicit memory, prevents adjacent (neighbor) points appearing in succession. The resulting distance between two successive points after convolution encoding is therefore bigger than the distance between any two points in Figure 2. This translates to a more accurate decision of the received state and hence better performance under high Gaussian noise.

We now consider the implementation of this modem encoder with convolution encoding. Figure 4 shows the block diagram of the full V.32 encoder with convolution encoding. Figure 5 shows the implementation of this circuit in two PAL devices, a PAL20X8A and a PAL20RS8. The quadrant-change state machine and the convolution encoder are implemented in the PAL20X8A and the coordinate mapping of the 32-point signal constellation is achieved through the PAL20RS8 device.

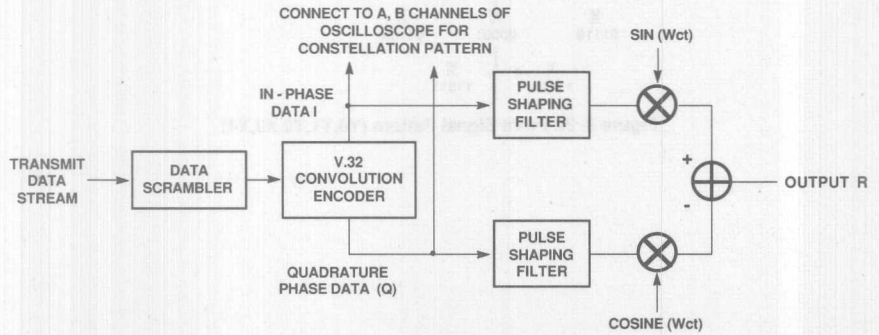


Figure 1. V.32 Convolution Encoder in a Typical Modem Transmitter

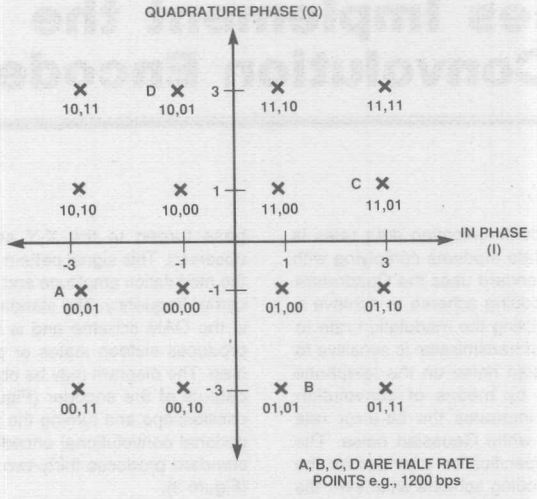


Figure 2. QAM Signal Pattern (Y1,Y2,X3,X4)

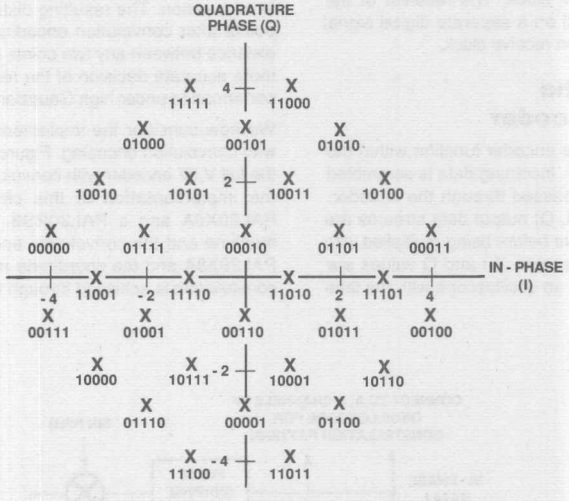


Figure 3. 32-Point Signal Pattern (Y0,Y1,Y2,X3,X4)

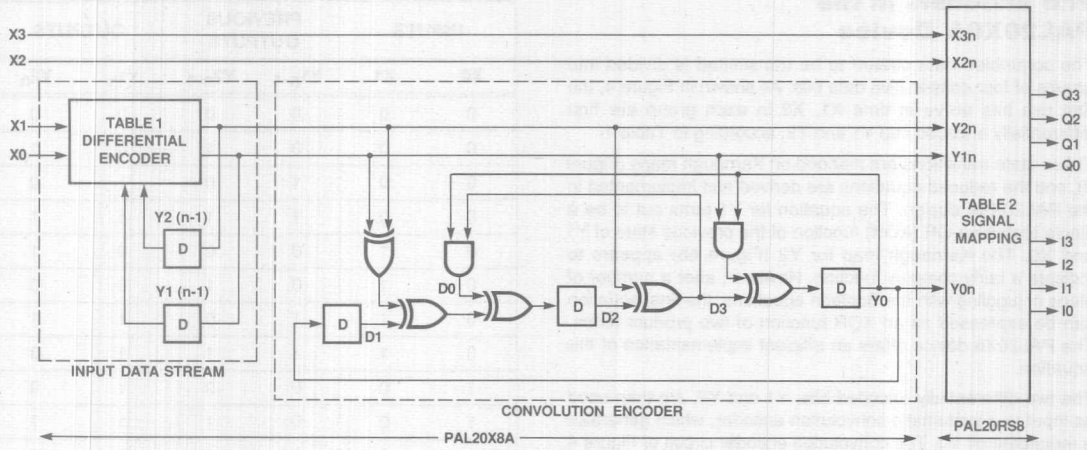


Figure 4. Full V.32 Convolution Encoder (9600 bps)

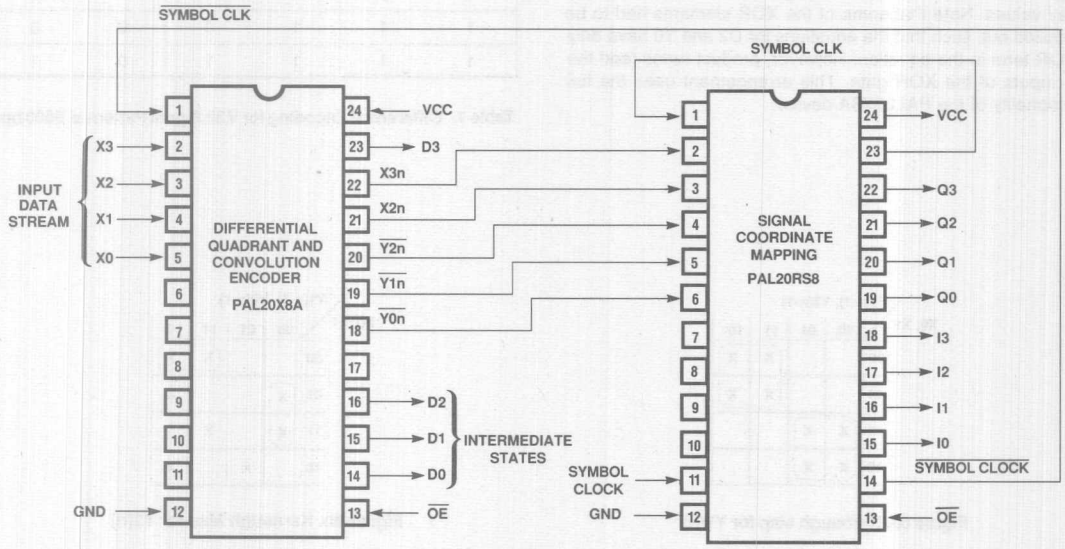


Figure 5. V.32 Convolution Encoder

PAL20X8A Device

The scrambled data stream to be transmitted is divided into groups of four consecutive data bits. As shown in Figure 4, the first two bits arrive in time X1, X2 in each group are first differentially encoded into Y1 and Y2, according to Table 1.

These state transitions are mapped on Karnaugh maps (Figure 6), and the reduced equations are derived and implemented in the PAL20X8A design. The equation for Y1 turns out to be a single Exclusive-OR (XOR) function of the previous state of Y1 and X0. The Karnaugh map for Y2 (Figure 6b) appears to indicate a fairly complex function. However, after a number of steps of juggling with the Boolean equations, the final equation can be expressed as an XOR function of two product terms. The PAL20X8 device offers an efficient implementation of this equation.

The two differentially encoded bits, Y1 and Y2, are then used as input to a systematic convolution encoder, which generates a redundant bit Y0. The convolution encoder circuit of Figure 4 contains five XOR functions and this requires the use of a PAL device containing XOR elements. The XORs feed into each other. Combinatorial values D0 and D3 are produced (see equations for pin 14 and 23 of the PAL20X8A). Additionally, registered values D1 and D2 are generated as intermediate delay values. Note that some of the XOR elements had to be multiplied out, such that the equations for D2 and Y0 have only 1 XOR term in the equation. However, product terms feed the two inputs of the XOR gate. This arrangement uses the full functionality of the PAL20X8A device.

INPUTS		PREVIOUS OUTPUTS		OUTPUTS	
X0	X1	$Y1_{n-1}$	$Y2_{n-1}$	$Y1_n$	$Y2_n$
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	1	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	1	1
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	1

Table 1. Differential Encoding for V.32 Signal Pattern at 9600 bps

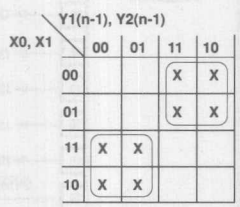


Figure 6a. Karnaugh Map for Y1(n)

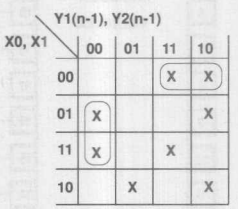


Figure 6b. Karnaugh Map for Y2(n)

Signal Mapping in PAL20RS8 Device

The extra bit Y0 and the four information carrying bits Y1, Y2, X2, and X3 are then mapped into the coordinates of the signal element to be transmitted according to the signal space diagram shown in Figure 3 and listed in Table 2. The coordinate values for I and Q are each represented as a 4-bit binary number. The values (I3,I2,I1,I0) 1100, 1011, 1010, 1001, 1000, 0111, 0110, 0101, 0100 represent the magnitudes +4, +3, +2, +1, 0, -1, -2, -3, -4 respectively. The values for the Q bits have a similar representation. Karnaugh maps (Figure 7) are produced for each of the outputs I3 to I0 and Q3 to Q0, and the reduced equations are implemented in the PAL device design. These eight output equations are dependent on only five outputs, Y0, Y1, Y2, X2, and X3, but one of the outputs (I2) requires nine product terms. A standard PAL device (e.g., PAL20R8) has only eight product terms. A PAL device (such as PAL20RS8) which has sixteen product terms shared between pairs of outputs would be fine. The PAL20RS8 is ideally suited to this application because it allows the nine product terms required for calculating I2 to be allocated to this output (on pin 17) and the other seven product terms to be allocated to the neighboring signal I3 (on pin 18).

Results

The output of the first PAL20X8A is available after the first half of the symbol clock and the final output is available from the second PAL20RS8 after the second half of the symbol clock. Thus a delay of only one symbol clock is introduced by the full encoder. The two PAL device designs have been verified and the 32-state constellation of Figure 3 was observed on an oscilloscope. The encoding rate was 2400 baud, but the PAL device design will work at 10 MHz for applications that may require this speed.

Summary

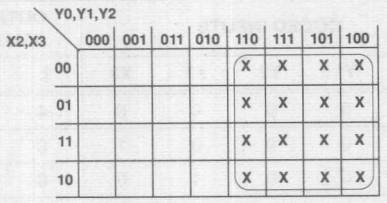
The full V.32 encoder with convolution encoding has been implemented in two PAL devices as shown in Figure 5. Both the differential quadrant encoder state machine and the convolution encoder fit into the PAL20X8A and the signal co-ordinate mapping is achieved through the PAL20RS8. This PAL device design for the encoder provides a superior way of implementing a V.32 modem transmitter synchronized only to the transmit clock.

References

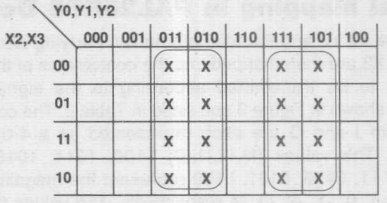
1. CCITT V series of recommendations, Red Book (1984), Volume VIII.1.

CODED INPUTS					OUTPUT VALUES	
(Y0)	Y1	Y2	X2	X3	I	Q
0	0	0	0	0	-4	1
	0	0	0	1	0	-3
	0	0	1	0	0	1
	0	0	1	1	4	1
	0	1	0	0	4	-1
	0	1	0	1	0	3
	0	1	1	0	0	-1
	0	1	1	1	-4	-1
	1	0	0	0	-2	3
	1	0	0	1	-2	-1
	1	0	1	0	2	3
	1	0	1	1	2	-1
	1	1	0	0	2	-3
	1	1	0	1	2	1
	1	1	1	0	-2	-3
	1	1	1	1	-2	1
1	0	0	0	0	-3	-2
	0	0	0	0	1	-2
	0	0	1	0	-3	2
	0	0	1	1	1	2
	0	1	0	0	3	2
	0	1	0	1	-1	2
	0	1	1	0	3	-2
	0	1	1	1	-1	-2
	1	0	0	0	1	4
	1	0	0	1	-3	0
	1	0	1	0	1	0
	1	0	1	1	1	-4
	1	1	0	0	-1	-4
	1	1	0	1	3	0
	1	1	1	0	-1	0
	1	1	1	1	-1	4

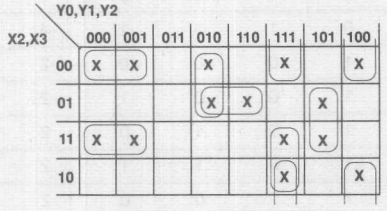
Table 2. Signal State Mapping for V.32, 9600 bps



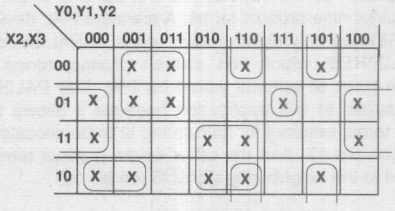
Karnaugh Map for I0



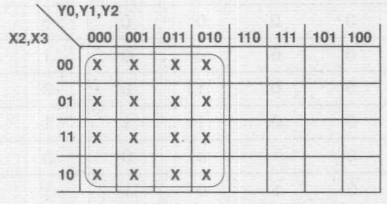
Karnaugh Map for I1



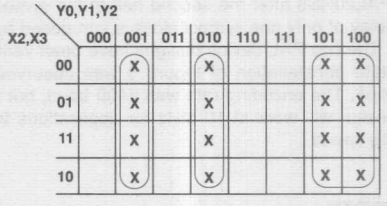
Karnaugh Map for I2



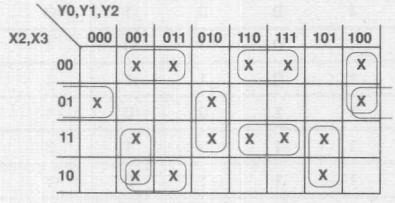
Karnaugh Map for I3



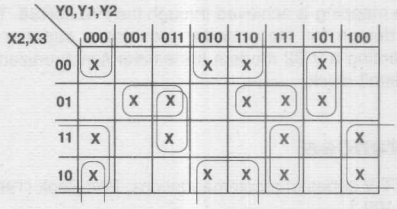
Karnaugh Map for Q0



Karnaugh Map for Q1



Karnaugh Map for Q2



Karnaugh Map for Q3

Figure 7. Karnaugh maps for outputs I0-I3 and Q0-Q3

```
;V.32 trellis encoder PAL#1, 20X8
;This PAL device performs the quadrant change based on the first
;two bits and the the convolution encoding to produce the output y0.
;

TITLE     V32_1.PDS
PATTERN   V32_ENCODER
REVISION A
AUTHOR    RAJ PARIPATYADAR
COMPANY   MMI, SANTA CLARA
DATE      APRIL 3 1987

CHIP  TREL1 PAL20X8

;PINS     1    2    3    4    5    6    7    8    9    10   11   12
          CK   X3   X2   X1   X0   NC   NC   NC   NC   NC   NC   GND
;PINS     13   14    15   16   17   18   19   20   21   22   23   24
         /OE  /D0   /D1  /D2  NC   /Y0n /Y1n /Y2n X2n  X3n  /D3  VCC

EQUATIONS

;These equations implement the differential quadrant encoder
;on the bits X0 and X1. The outputs are Y1n and Y2n.


Y1n      := Y1n :+: X0

Y2n      := (X0 *Y1n) :+:  (X1*/Y2n + /X1*Y2n)

;These equations produce the extra output Y0n.  Intermediate values
;D0,D1,D2 AND D3 need to be calculated first. D1 and D2 are
;registered outputs.


D1       := Y0n

D2       := (D0*/D1 + /D0*D1)
            :+: (Y1n*/Y2n + /Y1n*Y2n)

Y0n      := Y1n * Y0n :+: D3

D0       = Y0n*D2*/Y2n + Y0n*/D2*Y2n

D3       = D2*/Y2n + /D2*Y2n


;The following equations are just registered storage of the bits
;X2 and X3

/X2n     := /X2                    ;internal node equation

/X3n     := /X3                    ;internal node equation
```

```
TRACE_ON  X0  X1  Y2n  Y1n  Y0n  CK  D0  D1  D2  D3  X2n  X3n

SETF    OE                          ;Initialize Output-Enable
PRLDF   D1 D2 Y0n /Y1n /Y2n X2n X3n

SETF    X0  /X1  /X2  /X3           ;state Y1n=0 Y2n=0 with input
                                    ;PRELOAD VALUES ARE OPPOSITE
                                    ;TO THOSE INTENDED, TO GET AROUND
                                    ;A BUG IN PALASM2 V2.2
                                    ;check the state transitions from

CLOCKF                              ;X0=1  and X1=0
CLOCKF                              ;X2=0 and X3=0 sets the smallest
CLOCKF                              ;vector amplitude
CLOCKF                              ;output available on the negative
                                    ;edge of clock
;
;
SETF    X0  X1
PRLDF   D1 D2 Y0n /Y1n /Y2n X2n X3n
                                    ;Check the state transitions from
CLOCKF                              ;state Y1n=0  Y2n=0 with inputs
CLOCKF                              ;X0=1 and X1=1
CLOCKF
CLOCKF


SETF    /X0  X1
PRLDF   D1 D2 Y0n /Y1n /Y2n X2n X3n
                                    ;check the state transitions from
CLOCKF                              ;state Y1n=0 Y2n=0 with inputs
CLOCKF                              ;X0=0 and X1=1


PRLDF   D1 D2 Y0n Y1n /Y2n X2n X3n
                                    ;check the state transitions from
CLOCKF                              ;state Y1n=1 Y2n=0 with inputs
CLOCKF                              ;X0=0 and X1=1


SETF    /X0 /X1
PRLDF   D1 D2 Y0n /Y1n /Y2n X2n X3n
                                    ;check the state transition from
CLOCKF                              ;state Y1n=0 Y2n=0 with inputs
                                    ;X0=0 and X1=0

PRLDF   D1 D2 Y0n /Y1n Y2n X2n X3n
                                    ;check the state transitions from
CLOCKF                              ;state Y1n=0 Y2n=1 with inputs
                                    ;X0=0 and X1=0
```

```
PRLDF   D1 D2 Y0n Y1n /Y2n X2n X3n
                                ;check the state transitions from
CLOCKF                          ;state Y1n=1 Y2n=0 with inputs
                                ;X0=0 and X1=0

PRLDF   D1 D2 Y0n Y1n Y2n X2n X3n
                                ;check the state transitions from
CLOCKF                          ;state Y1n=1 Y2n=1 with inputs
                                ;X0=0 and X1=0

TRACE_OFF
```

```
;V.32 trellis encoder PAL#2, 20RS8.
;This PAL performs the signal  mapping onto the 32
;state constellation according to  CCITT V.32, 9600 bps
;specification.

TITLE    V32_ENCODER
PATTERN  V32_2.PDS
REVISION A
AUTHOR   RAJ PARIPATYADAR
COMPANY  MMI, SANTA CLARA
DATE     APRIL 1 1987

CHIP  TREL2 PAL20RS8

;PINS    1    2    3    4    5    6    7    8    9    10   11   12
         CLK  X3a  X2a  /Y2  /Y1  /Y0  NC   NC   NC   NC   SCKIN GND
;PINS    13   14   15   16   17   18   19   20   21   22   23   24
        /OE INVSCK I0   I1   I2   I3   Q0   Q1   Q2   Q3   SCK  VCC


EQUATIONS

I3      := /Y1*/Y2*X3a                          ;IN-PHASE OUTPUT .
        +  /Y0*/Y1*X2a*/X3a                     ;(I0--I3)
        +  /Y0*Y2*/X2a
        +  Y1*/Y2*X2a
        +  Y0*Y1*/Y2*/X3a
        +  Y0*Y1*Y2*/X2a*X3a
        +  Y0*/Y1*Y2*/X3a

I2      := /Y0*/Y1*/X2a*/X3a
        +  /Y0*/Y1*X2a*X3a
        +  /Y0*Y1*Y2*X2a
        +  /Y0*Y1*/Y2*/X2a
        +  Y1*/Y2*/X2a*X3a
        +  Y0*Y1*Y2*/X3a
        +  Y0*Y2*X2a*X3a
        +  Y0*/Y1*Y2*X3a
        +  Y0*/Y1*/Y2*/X3a


I1      := /Y0*Y1
        +  Y0*Y2

I0      := Y0


Q3      := /Y0*/Y1*/Y2*/X3a                     ;QUADRATURE OUTPUT
        +  /Y1*/Y2*X2a                          ;(Q0-Q3)
        +  /Y0*Y2*/X2a*X3a
        +  /Y0*Y1*Y2*X3a
        +  Y1*/Y2*/X3a
        +  Y0*Y1*/X2a*X3a
        +  Y0*Y1*Y2*X2a
```

```
        +  Y0*/Y1*Y2*/X2a

Q2      := /Y1*/Y2*/X2a*X3a
        +  /Y0*Y2*/X2a*/X3a
        +  /Y0*Y2*X2a*/X3a
        +  /Y1*Y2*X2a
        +  /Y0*Y1*/Y2*X3a
        +  Y0*Y1*/X2a*/X3a
        +  Y0*Y1*X2a*X3a
        +  Y0*/Y1*/Y2*/X2a


Q1      := /Y0*/Y1*Y2
        +  /Y0*Y1*/Y2
        +  Y0*/Y1

Q0      := /Y0


INVSCK  =  /SCKIN                          ;GENERATE /SYMBOL CLOCK

SCK     =  SCKIN                           ;GENERATE SYMBOL CLOCK

SIMULATION

TRACE_ON  Y0  Y1  Y2  X2a  X3a  CLK
          I0  I1  I2  I3   Q0   Q1  Q2  Q3

SETF  OE  /CLK


SETF    /Y0  /Y1  /Y2  /X2A  /X3A
CLOCKF
SETF    /Y0  /Y1  /Y2  /X2A   X3A
CLOCKF
SETF    /Y0  /Y1  /Y2   X2A  /X3A
CLOCKF
SETF    /Y0  /Y1  /Y2   X2A   X3A
CLOCKF
SETF    /Y0  /Y1   Y2  /X2A  /X3A
CLOCKF
SETF    /Y0  /Y1   Y2  /X2A   X3A
CLOCKF
SETF    /Y0  /Y1   Y2   X2A  /X3A
CLOCKF
SETF    /Y0  /Y1   Y2   X2A   X3A
CLOCKF
SETF    /Y0   Y1  /Y2  /X2A  /X3A
CLOCKF
SETF    /Y0   Y1  /Y2  /X2A   X3A
CLOCKF
SETF    /Y0   Y1  /Y2   X2A  /X3A
CLOCKF
SETF    /Y0   Y1  /Y2   X2A   X3A
CLOCKF
```

```
CLOCKF
SETF    /Y0   Y1   Y2  /X2A   X3A
CLOCKF
SETF    /Y0   Y1   Y2   X2A  /X3A
CLOCKF
SETF    /Y0   Y1   Y2   X2A   X3A
CLOCKF
SETF     Y0  /Y1  /Y2  /X2A  /X3A
CLOCKF
SETF     Y0  /Y1  /Y2  /X2A   X3A
CLOCKF
SETF     Y0  /Y1  /Y2   X2A  /X3A
CLOCKF
SETF     Y0  /Y1  /Y2   X2A   X3A
CLOCKF
SETF     Y0  /Y1   Y2  /X2A  /X3A
CLOCKF
SETF     Y0  /Y1   Y2  /X2A   X3A
CLOCKF
SETF     Y0  /Y1   Y2   X2A  /X3A
CLOCKF
SETF     Y0  /Y1   Y2   X2A   X3A
CLOCKF
SETF     Y0   Y1  /Y2  /X2A  /X3A
CLOCKF
SETF     Y0   Y1  /Y2  /X2A   X3A
CLOCKF
SETF     Y0   Y1  /Y2   X2A  /X3A
CLOCKF
SETF     Y0   Y1  /Y2   X2A   X3A
CLOCKF
SETF     Y0   Y1   Y2  /X2A  /X3A
CLOCKF
SETF     Y0   Y1   Y2  /X2A   X3A
CLOCKF
SETF     Y0   Y1   Y2   X2A  /X3A
CLOCKF
SETF     Y0   Y1   Y2   X2A   X3A
CLOCKF
TRACE_OFF
```

PLD Devices Implement 4B3T Line Transcoder

AN-175

Applications of 4B3T Line Coding

Line coding devices are required in line terminals and line repeaters to convert binary data into a proper coded format on the transmission lines and vice versa (see Figure 1).

4B3T line coding is used for the Integrated Services Digital Network (ISDN) in Germany. Also, 4B3T is used on T-carrier transmission lines for two 24-channel DS-1 signals (forty-eight channels). Because 4B3T coding decreases transmission requirements by twenty-five percent, transmission rates for two 24-channel DS-1 signals are only fifty percent more than one 24-channel DS-1 signals. The 4B3T is used on the T148 transmission line developed by ITT Telecommunication.

Bipolar Codes

Bipolar coding is a popular coding scheme over long transmission lines. Bipolar codes use three different voltages (+5 V, 0 V, and -5 V) to represent logic one (1) and logic zero (0) (see Figure 2a). Logic-one signals alternate between +5 V and -5 V to allow the mean DC level to be integrated to zero volts. Therefore, the bipolar code can compensate for DC wander on the transmission lines. However, the bipolar code does not take advantage of using three logic levels. Thus, binary bipolar code is called a pseudo-ternary code.

Ternary Codes

Ternary code takes advantage of all three logic levels. Each bit of the ternary code represents a positive logic one (+), a logic zero (0), or a negative logic one (-) (see Figure 2b). These three logic levels are mapped to the three voltage levels.

On the transmission lines, the power of three (3^n) codes are carried using the ternary code and the power of two (2^n) codes are carried using the binary code. For example, two bits of binary (0, 1) codes only provide four combinations of data (00, 01, 10, and 11). But two bits of ternary (-, 0, +) codes provide nine combinations of data (00, 0-, -0, --, -+, 0+, +0, +-, and ++). The ternary code is more efficient than the binary code because more information can be represented on the transmission lines. Three bipolar bits are capable of representing eight codes. For ternary code, only two bits are required to represent eight codes as shown in Table 1.

Comparison of Binary Codes and Ternary Codes

4B3T is a ternary code which maps four continuous binary bits into three ternary bits. To transmit twenty binary data bits, twenty bipolar bits are needed. But with ternary code only fifteen bits are needed. A twenty-five percent saving on bits is made using 4B3T line coding on the transmission lines. Without increasing the transmission rate, ternary coding carries more information than bipolar coding. Four binary bits on the computer system will run at the same rate as three ternary bits on the transmission lines using the 4B3T line coding. For example, a group of four binary bits in a 10-MHz computer system. Substituted to a group of three ternary bits at 7.5 MHz on a transmission line.

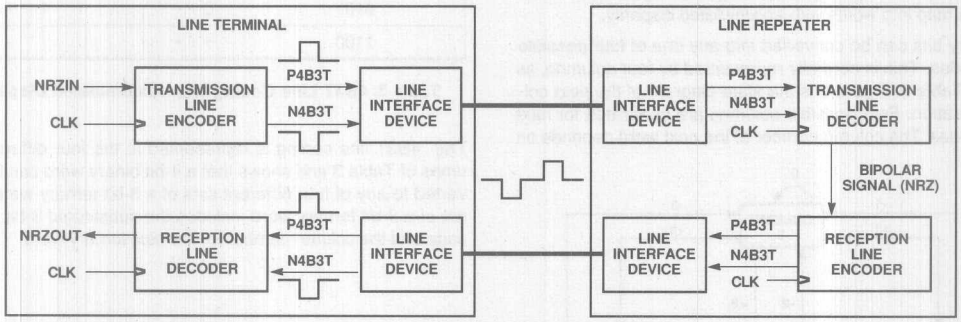


Figure 1. Line Coding Devices in a Line Terminal and a Line Repeater

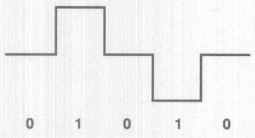


Figure 2a. Bipolar Codes

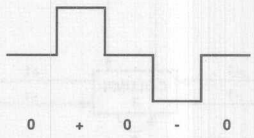


Figure 2b. Ternary Codes

CODE NUMBER	BINARY CODE (0, 1)			TERNARY CODE (-, 0, +)	
	B2	B1	B0	T1	T0
0	0	0	0	0	0
1	0	0	1	0	+
2	0	1	0	+	0
3	0	1	1	+	+
4	1	0	0	-	0
5	1	0	1	-	+
6	1	1	0	+	-
7	1	1	1	-	-

Table 1. Eight Codes Represented by Three Bipolar Bits or Two Ternary Bits

4B3T Line Coding

Four binary bits represent sixteen combinations of data, but three ternary bits represent twenty-seven combinations of data. Except for all zeros code, the remaining combinations of ternary codes can be categorized into three groups: zero, positive, and negative accumulated disparity. A zero-accumulated disparity is a set of ternary codes that have the same number of positive codes and negative codes, e.g., the accumulated disparity of 0-+ is zero (0). A positive accumulated disparity is a set of ternary codes that have more positive codes than negative codes, e.g., the accumulated disparity of the +-+ is positive (+1). A negative accumulated disparity is a set of ternary codes that have more negative codes than positive codes, e.g., the accumulated disparity of the 0-- is negative (-2). The 4B3T line coding with accumulated disparity is shown in Table 2. Six sets of four binary bits have zero-accumulated disparity. The remaining ten sets of four binary bits map into words with accumulated disparity.

Four binary bits can be converted into any one of four possible ternary codes. This is normally represented by four columns, as shown in Table 3. Figure 3 is the state diagram of the next column generation. Four possible columns are generated for next ternary codes. The column number of the next word depends on the previous column and the current accumulated disparity. If the current disparity is zero, the number of the next column is the same as the previous one. If the current disparity is positive, the column number for the next word is found by adding the current disparity. Likewise, if the current disparity is negative, the number for the the next column is found by subtracting the current disparity. In column 1, the next word may only have positive or zero disparity since the maximum of -3 negative disparity has been reached. Similarly, in column 4, the next word will have negative or zero disparity.

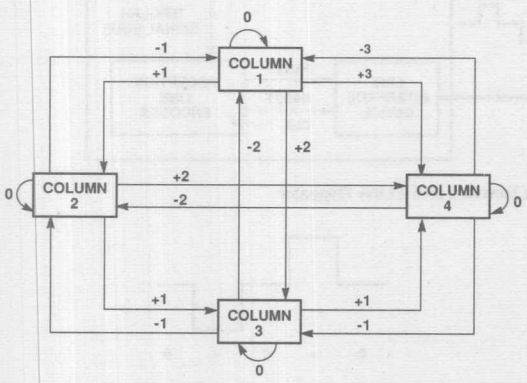


Figure 3. 4B3T Next Column Generation State Diagram

4-BIT BINARY WORD	3-BIT TERNARY WORD (ACCUMULATED DISPARITY)		
	POSITIVE	ZERO	NEGATIVE
0001		0 - +	
0111		- 0 +	
0100		- + 0	
0010		+ - 0	
1011		+ 0 -	
1110		0 + -	
1001	+ - +		- - -
0011	0 0 +		- - 0
1101	0 + 0		- 0 -
1000	+ 0 0		0 - -
0110	- + +		- - +
1010	+ + -		+ - -
1111	+ + 0		0 0 -
0000	+ 0 +		0 - 0
0101	0 + +		- 0 0
1100	+ + +		- + -

Table 2. 4B3T Line Coding with Accumulated Disparity

The 4B3T line coding is represented in the four different columns of Table 3 and shows that a 4-bit binary word can be converted to any of four different sets of a 3-bit ternary word. Each set of a 3-bit ternary word includes the substituted 3-bit ternary code and the column number for the next ternary word.

4-BIT BINARY WORD	COLUMN 1		COLUMN 2		COLUMN 3		COLUMN 4	
	3-BIT TERNARY WORD	*	3-BIT TERNARY WORD	*	3-BIT TERNARY WORD	*	3-BIT TERNARY WORD	*
0001	0 - +	1	0 - +	2	0 - +	3	0 - +	4
0111	- 0 +	1	- 0 +	2	- 0 +	3	- 0 +	4
0100	- + 0	1	- + 0	2	- + 0	3	- + 0	4
0010	+ - 0	1	+ - 0	2	+ - 0	3	+ - 0	4
1011	+ 0 -	1	+ 0 -	2	+ 0 -	3	+ 0 -	4
1110	0 + -	1	0 + -	2	0 + -	3	0 + -	4
1001	+ - +	2	+ - +	3	+ - +	4	- - -	1
0011	0 0 +	2	0 0 +	3	0 0 +	4	- - 0	2
1101	0 + 0	2	0 + 0	3	0 + 0	4	- 0 -	2
1000	+ 0 0	2	+ 0 0	3	+ 0 0	4	0 - -	2
0110	- + +	2	- + +	3	- - +	2	- - +	3
1010	+ + -	2	+ + -	3	+ - -	2	+ - -	3
1111	+ + 0	3	0 0 -	1	0 0 -	2	0 0 -	3
0000	+ 0 +	3	0 - 0	1	0 - 0	2	0 - 0	3
0101	0 + +	3	- 0 0	1	- 0 0	2	- 0 0	3
1100	+ + +	4	- + -	1	- + -	2	- + -	3

* Denotes next new column.

Table 3. 4B3T Line Coding

4B3T Line Encoder Implementation

A 4B3T encoder converts binary data into two signals: a positive ternary signal and a negative ternary signal. These two signals generate a three-voltage level signal for the transmission line. The 4B3T encoder divides the incoming binary data stream into 4-bit blocks. This 4-bit word is converted to a 3-bit ternary word. The 4-bit binary word and the number of the column generate a ternary word.

Figure 4 is the block diagram of the 4B3T encoder. It consists of a 4-bit serial-in parallel-out shift register, a 2-bit counter, a 6-bit input and an 8-bit output look-up table converter, and two 3-bit parallel-in serial-out shift registers. Two Programmable Array Logic (PAL) devices and one registered PROM device implement the 4B3T encoder (see Figure 5). The first PALC16R6Z performs a 4-bit serial-in parallel-out shift register and a 2-bit counter. A 4-bit shift register (SR3, SR2, SR1, and SR0) converts the serial data into a 4-bit parallel word. A 2-bit counter provides the code conversion's clock rate which is one-quarter the speed of the serial-in parallel-out shift register's clock rate. The 63RS481A implements the 6-input and 8-output look-up table converter. The 4B3T encoding conversion table is shown in Figure 6. The 4-bit binary word (B3, B2, B1, and B0) and 2-bit current column numbers (C1, C0) generate 8-bit output (PT2, PT1, PT0, NT2, NT1, NT0, NC1, and NC0). NC1 and NC0 are the next column numbers. They are fed into the 63RS481A as current column numbers for the next operation. The remaining six outputs are divided into two groups. One group is the positive ternary signals (PT2, PT1, and PT0). The second group is the negative ternary signals (NT2, NT1, and NT0). The second PALC16R6Z provides two 3-bit parallel-in serial-out shift registers at three-quarters external clock rate. PT2, PT1, and PT0 are serialized to generate the positive ternary signal (P4B3T). Likewise, NT2, NT1, and NT0 are serialized to produce the negative ternary signal (N4B3T).

4B3T Line Decoder Implementation

A 4B3T decoder detects three continuous pairs of positive ternary signals (P4B3T) and negative ternary signals (N4B3T) and converts them to 4-bit binary words. Figure 7 is the block diagram of the 4B3T decoder. It includes a 4B3T decoding state machine and an NRZ data generation.

A 5-bit state machine generates four inputs for the parallel-in serial-out shift register and a load signal. A PALC22V10 is used for the 4B3T decoding state machine because it has enough product terms and flexible outputs. The PALC22V10's output has programmable register bypass and programmable output polarity features. Q4 generates shift/load signal for the 4-bit parallel-in serial-out shift register (HCT 195). This shift/load signal will be assertive low only at every three clock cycles. The remaining four register outputs (Q3, Q2, Q1, and Q0) are parallel inputs for the 4-bit parallel-in serial-out shift register (see Figure 8). This shift register clocks at four-thirds external clock rate. The decoding state machine diagram is shown in Figure 9. An error flag is generated when positive ternary signals and negative ternary signals both are high. The error flag also detects three continuous pairs of zero ternary signals.

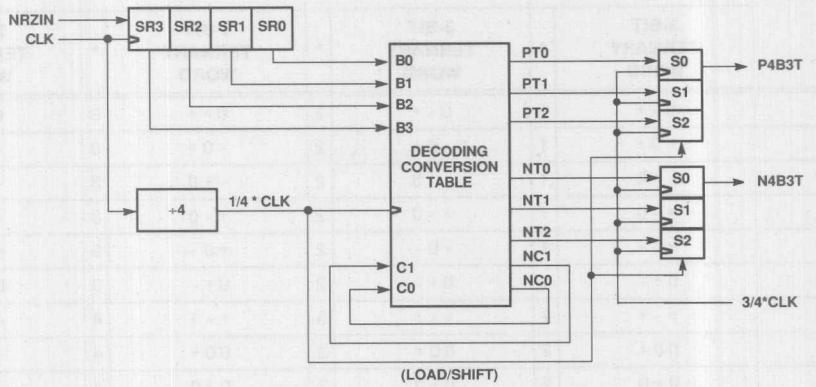


Figure 4. Block Diagram of 4B3T Line Encoding

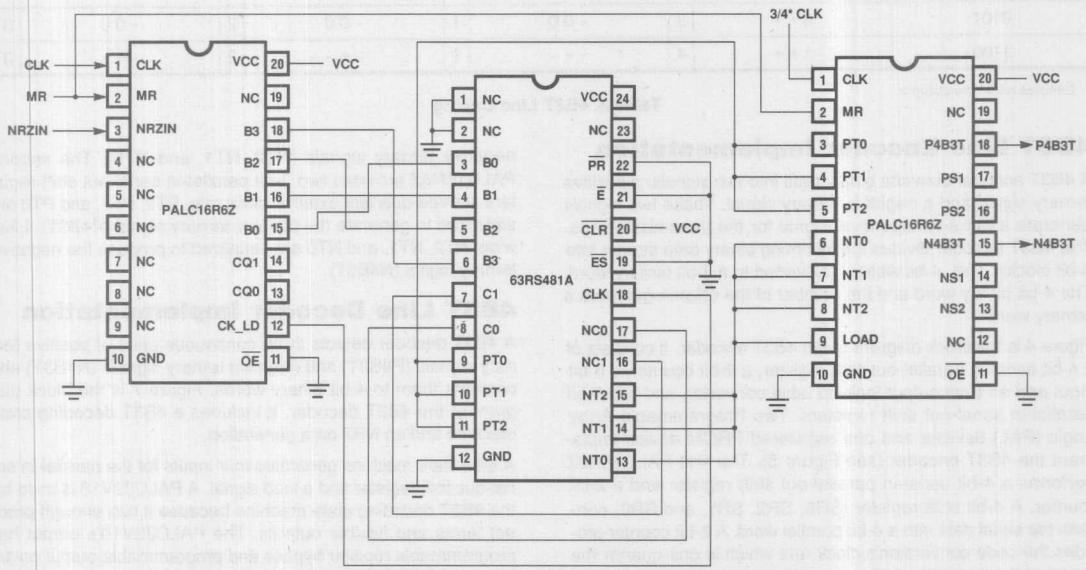


Figure 5. 4B3T Encoder Implemented by PLD Devices

B0 -- B3	C1 C0	T0 - T2	PT0 - PT2 and NT0 - NT2	NC1 NC0
0001	00 01 10 11	0 - +	001 and 010	00 01 10 11
0111	00 01 10 11	- 0 +	001 and 100	00 01 10 11
0100	00 01 10 11	- + 0	010 and 100	00 01 10 11
0010	00 01 10 11	+ - 0	100 and 010	00 01 10 11
1011	00 01 10 11	+ 0 -	100 and 001	00 01 10 11
1110	00 01 10 11	0+-	010 and 001	00 01 10 11
1001	00 01 10 11	+-+ +-+ +-+ ---	101 and 010 101 and 010 101 and 010 000 and 111	01 10 11 00
0011	00 01 10 11	00+ 00+ 00+ --0	001 and 000 001 and 000 001 and 000 000 and 110	01 10 11 01
1101	00 01 10 11	0 + 0 0 + 0 0 + 0 - 0 -	010 and 000 010 and 000 010 and 000 000 and 101	01 10 11 01
1000	00 01 10 11	+ 0 0 + 0 0 + 0 0 0 - -	100 and 000 100 and 000 100 and 000 000 and 011	01 10 11 01
0110	00 01 10 11	- + + - + + - - + - - +	011 and 100 011 and 100 001 and 110 001 and 110	01 10 01 10
1010	00 01 10 11	+ + - + + - + - - + - -	110 and 001 110 and 001 100 and 011 100 and 011	01 10 01 10
1111	00 01 10 11	+ + 0 0 0 - 0 0 - 0 0 -	110 and 000 000 and 001 000 and 001 000 and 001	10 00 01 10
0000	00 01 10 11	+ 0 + 0 - 0 0 - 0 0 - 0	101 and 000 000 and 010 000 and 010 000 and 010	10 00 01 10
0101	00 01 10 11	0 + + - 0 0 - 0 0 - 0 0	011 and 000 000 and 100 000 and 100 000 and 100	10 00 01 10
1100	00 01 10 11	+ + + - + - - + - - + -	111 and 000 010 and 101 010 and 101 010 and 101	11 00 01 10

Figure 6. 4B3T Encoding Conversion Table

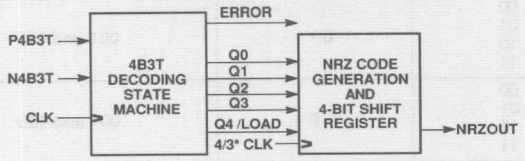


Figure 7. Block Diagram of 4B3T Line Decoding

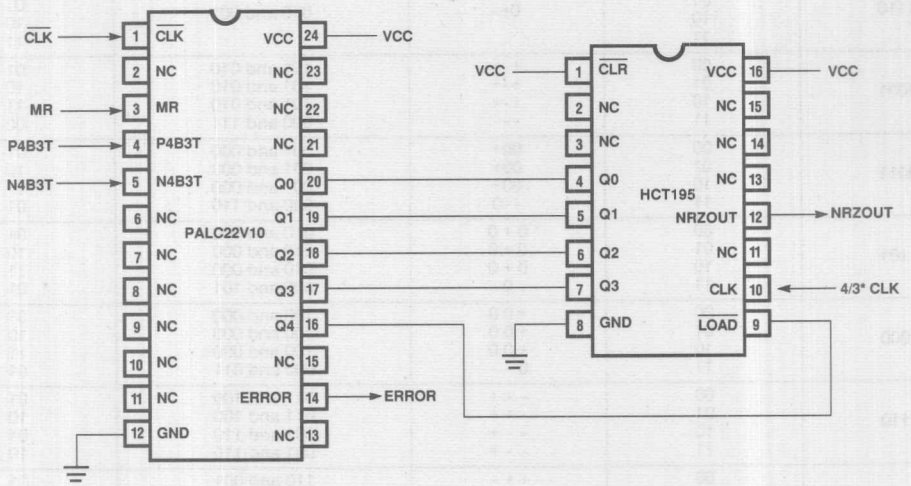


Figure 8. 4B3T Decoder Implemented by a PAL Device and a Shift Register

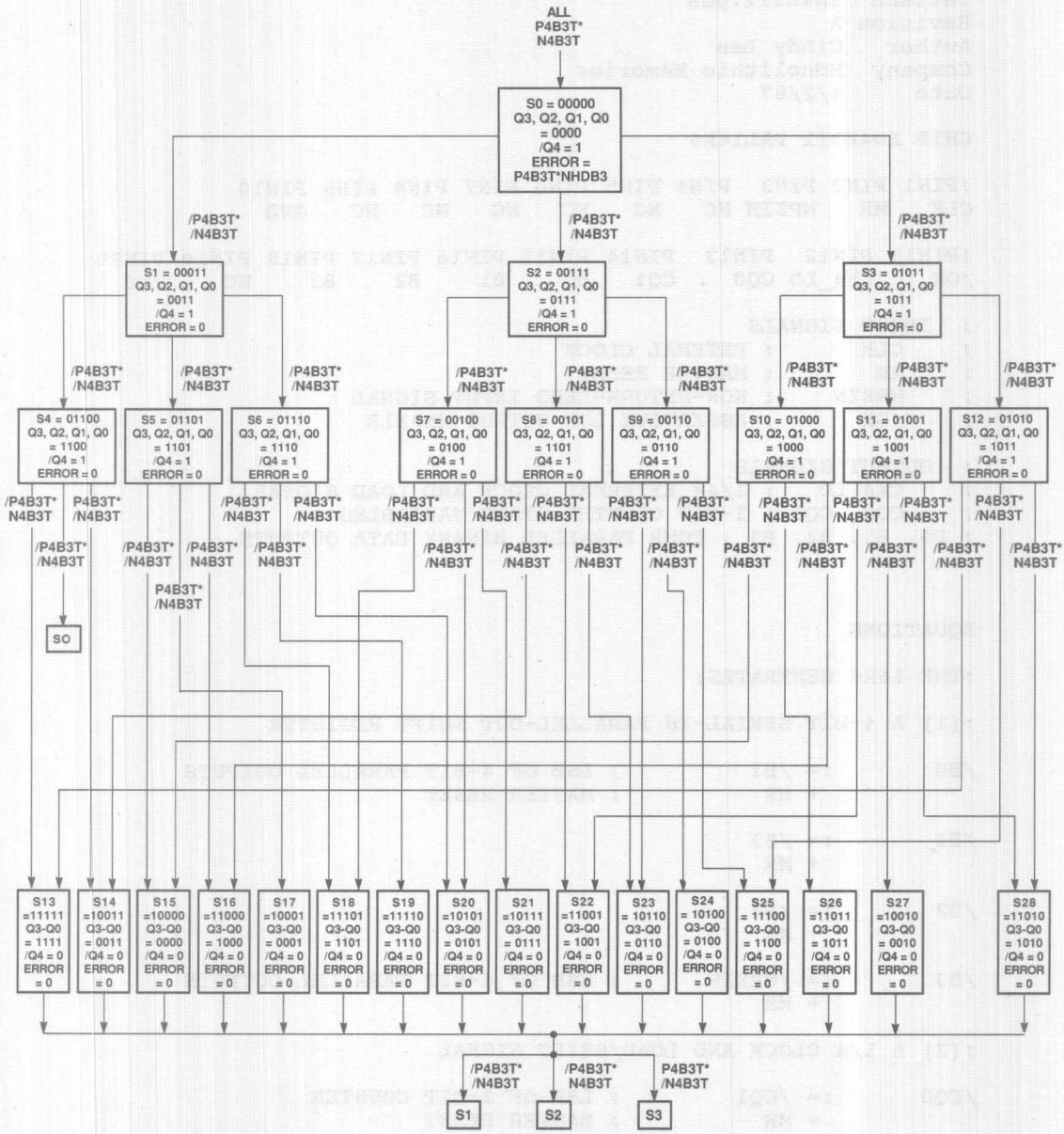


Figure 9. 4B3T Decoding State Machine

```
Revision A
Author   Cindy Lee
Company  Monolithic Memories
Date     4/2/87

CHIP EN4B3T1 PAL16R6

;PIN1 PIN2 PIN3  PIN4 PIN5 PIN6 PIN7 PIN8 PIN9 PIN10
CLK   MR   NRZIN NC   NC   NC   NC   NC   NC   GND

;PIN11 PIN12  PIN13  PIN14 PIN15 PIN16 PIN17 PIN18 PIN19 PIN20
/OE    CK4_LD CQ0    CQ1   B0    B1    B2    B3    NC    VCC

;  INPUT SIGNALS
;    CLK      : EXTENAL CLOCK
;    MR       : MASTER RESET
;    NRZIN    : NON-RETURN-ZERO INPUT SIGNAL
;    /OE      : ASSERTIVE LOW OUTPUT ENABLE

;  OUTPUT SIGNALS
;    CK4_LD   : 1/4* EXTERNAL CLOCK AND LOAD SIGNAL
;    CQ0, CQ1 : 2-BIT COUNTER STATE VARIABLES
; B0, B1, B2, B3 : FOUR PARALLEL BINARY DATA OUTPUTS


EQUATIONS

;THE 16R6 GENERATES:

;(1) A 4-BIT SERIAL-IN PARALLEL-OUT SHIFT REGISTER

/B0       := /B1          ; LSB OF 4-BIT PARALLEL OUTPUTS
           + MR           ; MASTER RESET

/B1       := /B2
           + MR

/B2       := /B3
           + MR

/B3       := /NRZIN       ; MSB OF 4-BIT PARALLEL OUTPUTS
           + MR

;(2) A 1/4 CLOCK AND LOAD/SHIFT SIGNAL

/CQ0      := /CQ1         ; LSB OF 2-BIT COUNTER
           + MR           ; MASTER RESET

/CQ1      := CQ0          ; MSB OF 2-BIT COUNTER
           + MR

/CK4_LD    = CQ0          ; 1/4 * EXTERNAL CLOCK
           + CQ1          ; AND A LOAD/SHIFT SIGNAL
```

```
SIMULATION

TRACE_ON  MR CLK NRZIN B0 B1 B2 B3 CQ0 CQ1 CK4_LD

SETF MR OE                ; RESET CONDITION
CLOCKF CLK

SETF NRZIN
CLOCKF CLK

SETF /MR NRZIN            ; DISABLE THE MASTER RESET
CLOCKF CLK                ; SHIFT A POSITIVE NRZ DATA

SETF /NRZIN               ; SHIFT TWO NEGATIVE NRZ DATA
CLOCKF CLK

SETF /NRZIN
CLOCKF CLK

SETF NRZIN                ; SHIFT TWO POSITIVE NRZ DATA
CLOCKF CLK

SETF NRZIN
CLOCKF CLK

SETF /NRZIN               ; SHIFT A NEGATIVE NRZ DATA
CLOCKF CLK

SETF NRZIN                ; SHIFT A POSITIVE NRZ DATA
CLOCKF CLK

SETF /NRZIN               ; SHIFT THREE NEGATIVE NRZ DATA
CLOCKF CLK

SETF /NRZIN
CLOCKF CLK

SETF /NRZIN
CLOCKF CLK

SETF NRZIN                ; SHIFT A POSITIVE NRZ DATA
CLOCKF CLK

SETF /NRZIN               ; SHIFT A NEGATIVE NRZ DATA
CLOCKF CLK

SETF NRZIN                ; SHIFT THREE POSITIVE NRZ DATA
CLOCKF CLK

SETF NRZIN
CLOCKF CLK

SETF NRZIN
CLOCKF CLK

TRACE_OFF
```

```
PLE9R8  ; PROM 63RS481A                          PLE DESIGN SPECIFICATION
TABLE.PLE                                        CINDY LEE 3/15/87
6-BIT TO 8-BIT TABLE CONVERTER
MMI, SANTA CLARA, CA

;PIN  8  7  6  5  4  3
;    A0 A1 A2 A3 A4 A5
;ADD C0 C1 B3 B2 B1 B0

;PIN  9  10  11  13  14  15  16  17
;    Q1  Q2  Q3  Q4  Q5  Q6  Q7  Q8
;DAT PT0 PT1 PT2 NT0 NT1 NT2 NC1 NC0


                                                    ; B0 B1 B2 B3 C1 C0
;-------------------------------------------------------------------

PT0 := /B0 * /B1 * /B2 * /B3 * /C1 * /C0    ; 0  0  0  0  0  0
     + /B0 * /B1 *  B2 * /B3                ; 0  0  1  0  X  X
     +  B0 * /B1                * /C1       ; 1  0  X  X  0  X
     +  B0 * /B1                *  C1 * /C0 ; 1  0  X  X  1  0
     +  B0 * /B1 *  B2          *  C1 *  C0 ; 1  0  1  X  1  1
     +  B0 *  B1 * /B2 * /B3 * /C1 * /C0    ; 1  1  0  0  0  0
     +  B0 *  B1 *  B2 *  B3 * /C1 * /C0    ; 1  1  1  1  0  0

PT1 :=       B1 * /B2 * /B3                 ; X  1  0  0  X  X
     + /B0 *  B1 * /B2 *  B3 * /C1 * /C0    ; 0  1  0  1  0  0
     + /B0 *  B1 *  B2 * /B3 * /C1          ; 0  1  1  0  0  X
     +  B0 * /B1 *  B2 * /B3 * /C1          ; 1  0  1  0  0  X
     +  B0 *  B1 * /B2 *  B3 * /C1          ; 1  1  0  1  0  X
     +  B0 *  B1 * /B2 *  B3 *  C1 * /C0    ; 1  1  0  1  1  0
     +  B0 *  B1 *  B2 * /B3                ; 1  1  1  0  X  X
     +  B0 *  B1 *  B2 *  B3 * /C1 * /C0    ; 1  1  1  1  0  0

PT2 := /B0 * /B1 * /B2 * /B3 * /C1 * /C0    ; 0  0  0  0  0  0
     + /B0 * /B1 * /B2 *  B3                ; 0  0  0  1  X  X
     + /B0 * /B1 *  B2 *  B3 * /C1          ; 0  0  1  1  0  X
     + /B0 * /B1 *  B2 *  B3 *  C1 * /C0    ; 0  0  1  1  1  0
     + /B0 *  B1 * /B2 *  B3 * /C1 * /C0    ; 0  1  0  1  0  0
     + /B0 *  B1 *  B2                      ; 0  1  1  X  X  X
     +  B0 * /B1 * /B2 *  B3 * /C1          ; 1  0  0  1  0  X
     +  B0 * /B1 * /B2 *  B3 *  C1 * /C0    ; 1  0  0  1  1  0
     +  B0 *  B1 * /B2 * /B3 * /C1 * /C0    ; 1  1  0  0  0  0

NT0 := /B0 * /B1 *  B2 *  B3 *  C1 *  C0    ; 0  0  1  1  1  1
     + /B0 *  B1 * /B2 * /B3                ; 0  1  0  0  X  X
     + /B0 *  B1 * /B2 *  B3 * /C1 *  C0    ; 0  1  0  1  0  1
     + /B0 *  B1 * /B2 *  B3 *  C1          ; 0  1  0  1  1  X
     + /B0 *  B1 *  B2                      ; 0  1  1  X  X  X
     +  B0       * /B2 *  B3 *  C1 *  C0    ; 1  X  0  1  1  1
     +  B0 *  B1 * /B2 * /B3 *  C1          ; 1  1  0  0  1  X
     +  B0 *  B1 * /B2 * /B3 * /C1 *  C0    ; 1  1  0  0  0  1

NT1 := /B0 * /B1 * /B2 * /B3 * /C1 *  C0    ; 0  0  0  0  0  1
     + /B0 * /B1 * /B2 * /B3 *  C1          ; 0  0  0  0  1  X
```

```
      +        /B1 * /B2 *  B3                          ; X  0  0  1  X  X
      + /B0 * /B1 *  B2 * /B3                           ; 0  0  1  0  X  X
      + /B0 * /B1 *  B2 *  B3 *  C1 *  C0               ; 0  0  1  1  1  1
      + /B0 *  B1 *  B2 * /B3 *  C1                     ; 0  1  1  0  1  X
      +  B0 * /B1 * /B2 * /B3 *  C1 *  C0               ; 1  0  0  0  1  1
      +  B0 * /B1 *  B2 * /B3 *  C1                     ; 1  0  1  0  1  X

NT2 :=  B0 * /B1 * /B2         *  C1 *  C0               ; 1  0  0  X  1  1
      +  B0 * /B1 *  B2                                 ; 1  0  1  X  X  X
      +  B0 *  B1 * /B2 * /B3 * /C1 *  C0               ; 1  1  0  0  0  1
      +  B0 *  B1 * /B2 * /B3 *  C1                     ; 1  1  0  0  1  X
      +  B0 *  B1 * /B2 *  B3 *  C1 *  C0               ; 1  1  0  1  1  1
      +  B0 *  B1 *  B2 * /B3                           ; 1  1  1  0  X  X
      +  B0 *  B1 *  B2 *  B3 * /C1 *  C0               ; 1  1  1  1  0  1
      +  B0 *  B1 *  B2 *  B3 *  C1                     ; 1  1  1  1  1  X

NC1 := /B0 * /B1 * /B2 *  B3 *  C1                     ; 0  0  0  1  1  0
      + /B0 * /B1 *  B2 * /B3 *  C1                     ; 0  0  1  0  1  X
      + /B0 *  B1 * /B2 * /B3 *  C1                     ; 0  1  0  0  1  X
      + /B0 *  B1 *  B2 *  B3 *  C1                     ; 0  1  1  1  1  X
      +  B0 * /B1 *  B2 *  B3 *  C1                     ; 1  0  1  1  1  X
      +  B0 *  B1 *  B2 * /B3 *  C1                     ; 1  1  1  0  1  X
      + /B0 * /B1 * /B2 * /B3 * /C1 * /C0               ; 0  0  0  0  0  0
      + /B0 * /B1 * /B2 * /B3 *  C1 *  C0               ; 0  0  0  0  1  1
      + /B0 * /B1 *  B2 *  B3 * /C1 *  C0               ; 0  0  1  1  0  1
      + /B0 * /B1 *  B2 *  B3 *  C1 * /C0               ; 0  0  1  1  1  0
      + /B0 *  B1 * /B2 *  B3 * /C1 * /C0               ; 0  1  0  1  0  0
      + /B0 *  B1 * /B2 *  B3 *  C1 *  C0               ; 0  1  0  1  1  1
      + /B0 *  B1 *  B2 * /B3         *  C0             ; 0  1  1  0  X  1
      +  B0 * /B1 * /B2 * /B3 *  C1 * /C0               ; 1  0  0  0  1  0
      +  B0 * /B1 * /B2 * /B3 * /C1 *  C0               ; 1  0  0  0  0  1
      +  B0         * /B2 *  B3 * /C1 *  C0             ; 1  X  0  1  0  1
      +  B0         * /B2 *  B3 *  C1 * /C0             ; 1  X  0  1  1  0
      +  B0 * /B1 *  B2 * /B3         *  C0             ; 1  0  1  0  X  1
      +  B0 *  B1 * /B2 * /B3 * /C1 * /C0               ; 1  1  0  0  0  0
      +  B0 *  B1 * /B2 * /B3 *  C1 *  C0               ; 1  1  0  0  1  1
      +  B0 *  B1 *  B2 *  B3 * /C1 * /C0               ; 1  1  1  1  0  0
      +  B0 *  B1 *  B2 *  B3 *  C1 *  C0               ; 1  1  1  1  1  1

NC0 := /B0 * /B1 * /B2 *  B3         *  C0             ; 0  0  0  1  X  1
      + /B0 * /B1 *  B2 * /B3         *  C0             ; 0  0  1  0  X  1
      + /B0 *  B1 * /B2 * /B3         *  C0             ; 0  1  0  0  X  1
      + /B0 *  B1 *  B2 *  B3         *  C0             ; 0  1  1  1  X  1
      +  B0 * /B1 *  B2 *  B3         *  C0             ; 1  0  1  1  X  1
      +  B0 *  B1 *  B2 * /B3         *  C0             ; 1  1  1  0  X  1
      + /B0 * /B1 * /B2 * /B3 *  C1 * /C0               ; 0  0  0  0  1  0
      + /B0 * /B1 *  B2 *  B3         * /C0             ; 0  0  1  1  X  0
      + /B0 * /B1 *  B2 *  B3 *  C1 *  C0               ; 0  0  1  1  1  1
      + /B0 *  B1 * /B2 *  B3 *  C1 * /C0               ; 0  1  0  1  1  0
      + /B0 *  B1 *  B2 * /B3         * /C0             ; 0  1  1  0  X  0
      +  B0 * /B1 * /B2 * /B3         * /C0             ; 1  0  0  0  X  0
      +  B0 * /B1 * /B2 * /B3 *  C1                     ; 1  0  0  0  1  X
      +  B0         * /B2 *  B3         * /C0           ; 1  X  0  1  X  0
      +  B0 * /B1 *  B2 * /B3         * /C0             ; 1  0  1  0  X  0
      +  B0 *  B1 * /B2 * /B3         * /C0             ; 1  1  0  0  X  0
      +  B0 *  B1 * /B2 *  B3 *  C1 *  C0               ; 1  1  0  1  1  1
      +  B0 *  B1 *  B2 *  B3 *  C1 * /C0               ; 1  1  1  1  1  0
```

```
B0 B1 B2 B3 C1 C0  PT0 PT1 PT2 NT0 NT1 NT2 NC1 NC0

;BINARY INPUT, COLUMN,    OUTPUT     NEW COLUMN
;B0 B1 B2 B3 C1 C0,     PT0 PT1 PT2 NT0 NT1 NT2 NC1 NC0
-------------------------------------------------------
  L  L  L  L  L  L       H   L   H   L   L   L   H   L
  L  L  L  L  L  H       L   L   L   L   H   L   L   L
  L  L  L  L  H  L       L   L   L   L   H   L   L   H
  L  L  L  L  H  H       L   L   L   L   H   L   H   L
;
  L  L  L  H  L  L       L   L   H   L   H   L   L   L
  L  L  L  H  L  H       L   L   H   L   H   L   L   H
  L  L  L  H  H  L       L   L   H   L   H   L   H   L
  L  L  L  H  H  H       L   L   H   L   H   L   H   H
;
  L  L  H  L  L  L       H   L   L   L   H   L   L   L
  L  L  H  L  L  H       H   L   L   L   H   L   L   H
  L  L  H  L  H  L       H   L   L   L   H   L   H   L
  L  L  H  L  H  H       H   L   L   L   H   L   H   H
;
  L  L  H  H  L  L       L   L   H   L   L   L   L   H
  L  L  H  H  L  H       L   L   H   L   L   L   H   L
  L  L  H  H  H  L       L   L   H   L   L   L   H   H
  L  L  H  H  H  H       L   L   L   H   H   L   L   H
;
  L  H  L  L  L  L       L   H   L   H   L   L   L   L
  L  H  L  L  L  H       L   H   L   H   L   L   L   H
  L  H  L  L  H  L       L   H   L   H   L   L   H   L
  L  H  L  L  H  H       L   H   L   H   L   L   H   H
;
  L  H  L  H  L  L       L   H   H   L   L   L   H   L
  L  H  L  H  L  H       L   L   L   H   L   L   L   L
  L  H  L  H  H  L       L   L   L   H   L   L   L   H
  L  H  L  H  H  H       L   L   L   H   L   L   H   L
;
  L  H  H  L  L  L       L   H   H   H   L   L   L   H
  L  H  H  L  L  H       L   H   H   H   L   L   H   L
  L  H  H  L  H  L       L   L   H   H   H   L   L   H
  L  H  H  L  H  H       L   L   H   H   H   L   H   L
;
  L  H  H  H  L  L       L   L   H   H   L   L   L   L
  L  H  H  H  L  H       L   L   H   H   L   L   L   H
  L  H  H  H  H  L       L   L   H   H   L   L   H   L
  L  H  H  H  H  H       L   L   H   H   L   L   H   H
;
  H  L  L  L  L  L       H   L   L   L   L   L   L   H
  H  L  L  L  L  H       H   L   L   L   L   L   H   L
  H  L  L  L  H  L       H   L   L   L   L   L   H   H
  H  L  L  L  H  H       L   L   L   L   H   H   L   H
;
```

```
  H   L   L   H   H   L       H   L   H   L   H   L   H   H
  H   L   L   H   H   H       L   L   L   H   H   H   L   L
;
  H   L   H   L   L   L       H   H   L   L   L   H   L   H
  H   L   H   L   L   H       H   H   L   L   L   H   H   L
  H   L   H   L   H   L       H   L   L   L   H   H   L   H
  H   L   H   L   H   H       H   L   L   L   H   H   H   L
;
  H   L   H   H   L   L       H   L   L   L   L   H   L   L
  H   L   H   H   L   H       H   L   L   L   L   H   L   H
  H   L   H   H   H   L       H   L   L   L   L   H   H   L
  H   L   H   H   H   H       H   L   L   L   L   H   H   H
;
  H   H   L   L   L   L       H   H   H   L   L   L   H   H
  H   H   L   L   L   H       L   H   L   H   L   H   L   L
  H   H   L   L   H   L       L   H   L   H   L   H   L   H
  H   H   L   L   H   H       L   H   L   H   L   H   H   L
;
  H   H   L   H   L   L       L   H   L   L   L   L   L   H
  H   H   L   H   L   H       L   H   L   L   L   L   H   L
  H   H   L   H   H   L       L   H   L   L   L   L   H   H
  H   H   L   H   H   H       L   L   L   H   L   H   L   H
;
  H   H   H   L   L   L       L   H   L   L   L   H   L   L
  H   H   H   L   L   H       L   H   L   L   L   H   L   H
  H   H   H   L   H   L       L   H   L   L   L   H   H   L
  H   H   H   L   H   H       L   H   L   L   L   H   H   H
;
  H   H   H   H   L   L       H   H   L   L   L   L   H   L
  H   H   H   H   L   H       L   L   L   L   L   H   L   L
  H   H   H   H   H   L       L   L   L   L   L   H   L   H
  H   H   H   H   H   H       L   L   L   L   L   H   H   L
```

DESCRIPTION

THIS REGISTERED PROM (63RS481A) IMPLEMENTS A 6-TO-8 TABLE CONVERTER. SIX INPUTS ARE B0, B1, B2, B3 (BINARY INPUTS), AND C1, C0 (CURRENT COLUMN NUMBER). BECAUSE THESE FOUR PARALLEL BINARY INPUTS ARE FROM THE 4-BIT SERIAL-IN PARALLEL-OUT SHIFT REGISTER, THIS PROM (63RS481A) MUST OPERATE AT A QUARTER OF THE SHIFT REGISTER'S RATE. EIGHT OUTPUTS ARE PT0, PT1, PT2, NT0, NT1, NT2, NC1, AND NC0. THE PT0, PT1, AND PT2 ARE POSITIVE TERNARY SIGNALS WHICH FORMS THREE SERIAL P4B3T SIGNAL THROUGH A 3-BIT PARALLEL-IN SERIAL-OUT SHIFT REGISTER. LIKEWISE, NT0, NT1, AND NT2 ARE NEGATIVE TERNARY SIGNALS. NC1 AND NC0 ARE NEXT COLUMN NUMBERS.

```
Title     4B3T ENCODER LOGIC2
Pattern   EN4B3T2.pds
Revision  A
Author    Cindy Lee
Company   Monolithic Memories
Date      3/5/87

CHIP EN4B3T2 PAL16R6

;PIN1 PIN2 PIN3 PIN4 PIN5 PIN6 PIN7 PIN8 PIN9 PIN1C
CLK   MR   PT0  PT1  PT2  NT0  NT1  NT2  LOAD GND

;PIN11 PIN12 PIN13 PIN14 PIN15 PIN16 PIN17 PIN18 PIN19 PIN20
/OE    NC    NS2   NS1   N4B3T PS2   PS1   P4B3T NC    VCC

;  INPUT SIGNALS
;     CLK     : EXTERNAL CLOCK
;     MR      : MASTER RESET
; PT0, PT1, PT2   : THREE POSITIVE TERNARY SIGNALS
; NT0, NT1, NT2   : THREE NEGATIVE TERNARY SIGNALS
;     LOAD    : LOAD SIGNAL FOR TWO SHIFT REGISTERS
;     /OE     : ASSERTIVE LOW OUTPUT ENABLE

;  OUTPUT SIGNALS
;  NS2, NS1 : NEGATIVE TERNARY SHIFT REGISTER VARIABLES
;     N4B3T   : OUTPUT SIGNAL OF NEGATIVE TERNARY SHIFT REGISTER
;   PS2, PS1 : NPOSITIVE TERNARY SHIFT REGISTER VARIABLES
;     P4B3T   : OUTPUT SIGNAL OF POSITIVE TERNARY SHIFT REGISTER

EQUATIONS

;THE 16R6 GENERATES:
;(1) A 3-BIT PARALLEL-IN SERIAL-OUT SHIFT REGISTER
;    FOR POSITIVE TERNARY SIGNAL

/P4B3T    := /PT0 * /PS1        ; SHIFT REGISTER OUTPUT
           + /LOAD * /PS1
           + /PT0 * LOAD
           + MR

/PS1      := /PT1 * /PS2
           + /LOAD * /PS2
           + /PT1 * LOAD
           + MR

/PS2      := /PT2
           + /LOAD
           + MR

;(2) A 3-BIT PARALLEL-IN SERIAL-OUT SHIFT REGISTER
;    FOR NEGATIVE TERNARY SIGNAL

/N4B3T    := /NT0 * /NS1        ; SHIFT REGISTER OUTPUT
           + /LOAD * /NS1
```

```
          + /NT0 * LOAD
          + MR

/NS1      := /NT1 * /NS2
          + /LOAD * /NS2
          + /NT1 * LOAD
          + MR

/NS2      := /NT2
          + /LOAD
          + MR


SIMULATION

TRACE_ON  MR CLK LOAD PT2 PT1 PT0 P4B3T NT2 NT1 NT0 N4B3T

SETF /LOAD MR OE                          ; RESET CONDITION
SETF /MR                                ; DISABLE MASTER RESET

SETF LOAD PT2 PT1 PT0 NT2 NT1 NT0       ; LOAD INPUT DATA
CLOCKF

SETF /LOAD /PT2 /PT1 PT0 /NT2 /NT1 NT0 ; SHIFT DATA
CLOCKF

SETF PT2 /PT1 PT0 NT2 /NT1 NT0          ; SHIFT DATA
CLOCKF

SETF LOAD PT2 /PT1 PT0 NT2 /NT1 NT0     ; LOAD INPUT DATA
CLOCKF

SETF /LOAD PT2 /PT1 PT0 NT2 /NT1 NT0    ; SHIFT DATA
CLOCKF

SETF /PT2 PT1 /PT0 NT2 NT1 /NT0         ; SHIFT DATA
CLOCKF

SETF LOAD PT2 /PT1 PT0 /NT2 NT1 NT0     ; LOAD INPUT DATA
CLOCKF

SETF /LOAD PT2 /PT1 PT0 /NT2 NT1 NT0    ; SHIFT DATA
CLOCKF

SETF PT2 /PT1 PT0 /NT2 NT1 NT0          ; SHIFT DATA
CLOCKF

SETF LOAD PT2 PT1 /PT0 /NT2 /NT1 /NT0   ; LOAD INPUT DATA
CLOCKF

SETF /LOAD PT2 /PT1 /PT0 NT2 /NT1 /NT0 ; SHIFT DATA
CLOCKF

SETF PT2 /PT1 /PT0 NT2 /NT1 /NT0        ; SHIFT DATA
CLOCKF
```

```
CLOCKF

SETF /LOAD PT2 /PT1 /PT0 NT2 /NT1 /NT0 ; SHIFT DATA
CLOCKF

SETF PT2 /PT1 /PT0 NT2 /NT1 /NT0        ; SHIFT DATA
CLOCKF

TRACE_OFF
```

```
Pattern   DE4B3T.pds
Revision A
Author    Cindy Lee
Company   Monolithic Memories
Date      3/18/87

CHIP DE4B3T PAL22V10

;PIN1 PIN2 PIN3 PIN4  PIN5  PIN6
/CLK  NC   MR   P4B3T N4B3T NC

;PIN7 PIN8 PIN9 PIN10 PIN11 PIN12
NC    NC   NC   NC    NC    GND

;PIN13 PIN14 PIN15 PIN16 PIN17 PIN18
NC     ERROR NC    /Q4   Q3    Q2

;PIN19 PIN20 PIN21 PIN22 PIN23 PIN24
Q1     Q0    NC    NC    NC    VCC
GLOBAL


;  INPUT SIGNALS
;    /CLK  : ASSERTIVE LOW EXTERNAL CLOCK
;    MR    : MASTER RESET
;    P4B3T : POSITIVE TERNARY SIGNAL
;    N4B3T : NEGATIVE TERNARY SIGNAL

;  OUTPUT SIGNALS
;    ERROR : ERROR FLAG
;    /Q4   : ASSERTIVE LOW Q4 SIGNAL WHICH IS A /LOAD SIGNAL
;            FOR A SHIFT REGISTER
; Q3, Q2, Q1, Q0 : 4B3T DECODING STATE VARIABLES


EQUATIONS

;ERROR SIGNAL IS GENERATED WHEN P4B3T AND N4B3T SIGNALS BOTH
;ARE HIGH, OR P4B3T AND N4B3T SIGNALS BOTH ARE LOW AT THE
;TRANSITION FROM STATE4 TO STATE0.

ERROR      = P4B3T * N4B3T              ; P4B3T AND N4B3T ARE HIGH
           + /Q4 * /Q3 * /Q2 * /Q1 * /Q0 * /P4B3T * /N4B3T

;/Q4 IA A ASSERTIVE LOW LOAD SIGNAL. THE FOUR PARALLEL DATA
;ARE LOADED INTO 4-BIT PARALLEL-IN SERIAL-OUT SHIFT REGISTER
;WHEN LAOD IS LOW, OTHERWISE, THE DATA IS SHIFTED OUT.

Q4         := /N4B3T * P4B3T * /Q1 * Q3 * /Q4     ; LOAD SIGNAL
            + /N4B3T * /Q0 * Q2 * /Q3 * /Q4
            + /N4B3T * /Q0 * Q1 * Q3 * /Q4
            + /N4B3T * /Q1 * Q2 * /Q3 * /Q4
            + N4B3T * /P4B3T * /Q1 * Q3 * /Q4
```

```
            + /P4B3T * /Q0 * Q2 * /Q3 * /Q4
            + /P4B3T * /Q0 * Q1 * Q3 * /Q4
            + /P4B3T * /Q1 * /Q2 * Q3 * /Q4
            + /P4B3T * /Q1 * Q2 * /Q3 * /Q4
            + /P4B3T * Q0 * /Q1 * Q3 * /Q4
GLOBAL.RSTF = MR

;Q3, Q2, Q1, AND Q0 ARE THE FOUR BINARY SIGNALS WHICH ARE
;CONVERTED FROM THTREE TERNARY CODE WORD.

Q3          := /N4B3T * P4B3T * /Q0 * /Q1 * /Q2 * /Q3
            + /N4B3T * P4B3T * Q0 * /Q2 * Q3     ; MSB OF THE FOUR
            + /N4B3T * Q0 * Q1 * /Q2 * /Q4       ; BINARY SIGNALS
            + /N4B3T * Q1 * /Q2 * Q3 * /Q4
            + /N4B3T * P4B3T * Q4
            + /P4B3T * /Q0 * /Q2 * Q3 * /Q4
            + N4B3T * /P4B3T * /Q0 * Q2 * /Q4
            + /P4B3T * /Q0 * Q1 * Q3 * /Q4
            + N4B3T * /P4B3T * /Q1 * Q2 * /Q4
            + N4B3T * /P4B3T * /Q1 * Q3 * /Q4
            + /P4B3T * Q0 * Q1 * /Q2 * /Q4


Q2          := /N4B3T * /Q0 * Q1 * Q3 * /Q4
            + /N4B3T * P4B3T * Q2 * /Q3 * /Q4
            + /N4B3T * Q0 * Q1 * /Q3 * /Q4
            + N4B3T * /P4B3T * /Q0 * /Q1 * /Q3
            + /P4B3T * /Q0 * Q2 * /Q3 * /Q4
            + N4B3T * /P4B3T * /Q0 * Q2
            + /P4B3T * Q0 * Q1 * /Q3 * /Q4
            + N4B3T * /P4B3T * Q4

Q1          := /N4B3T * /Q0 * /Q1 * /Q2 * /Q3
            + /N4B3T * P4B3T * /Q0 * /Q1 * Q2
            + /N4B3T * Q0 * /Q1 * Q2 * /Q3
            + /N4B3T * P4B3T * Q0 * Q1 * /Q2
            + /N4B3T * P4B3T * Q2 * /Q3
            + /N4B3T * Q4
            + /P4B3T * /Q0 * /Q1 * /Q2 * /Q3
            + /P4B3T * /Q0 * Q1 * /Q2 * Q3
            + N4B3T * /P4B3T * /Q0 * Q3
            + /P4B3T * Q0 * /Q1 * /Q2 * Q3
            + /P4B3T * Q4


Q0          := /N4B3T * /P4B3T * /Q0 * Q1 * Q3   ; LSB OF THE FOUR
            + /N4B3T * /Q0 * /Q1 * /Q3           ; BINARY SIGNALS
            + /N4B3T * /Q0 * Q1 * Q2 * Q3
            + /N4B3T * P4B3T * Q0 * /Q1 * Q3
            + /N4B3T * P4B3T * /Q1 * Q2 * Q3
            + /N4B3T * Q4
            + N4B3T * /P4B3T * /Q0 * /Q1
            + /P4B3T * /Q1 * Q2 * /Q3
            + N4B3T * /P4B3T * Q0 * Q1 * /Q2
            + N4B3T * /P4B3T * Q0 * Q2 * /Q3
            + /P4B3T * Q4
```

```
SIMULATION

TRACE_ON  MR /CLK P4B3T N4B3T /Q4 Q3 Q2 Q1 Q0 ERROR

;TO RESET WHOLE DEVICES

SETF MR                    ;STATE # = Q4(LOAD) Q3 Q2 Q1 Q0 & ERROR

;TEST VECTOR #1 TO CHECK S0, S1, S4, S13, AND S16

SETF /MR P4B3T N4B3T       ;S0 = 0 0 0 0 0, ERROR
CLOCKF CLK                 ;INITIAL CONDITION

SETF /P4B3T /N4B3T         ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S4 = 0 1 1 0 0
CLOCKF CLK

SETF /P4B3T N4B3T          ;S13 = 1 1 1 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S4 = =0 1 1 0 0
CLOCKF CLK

SETF P4B3T /N4B3T          ;S14 = 1 0 0  1 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S4 = 0 1 1 0 0
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S0 = 0 0 0 0 0, ERROR
CLOCKF CLK

;TEST VECTOR #2 TO CHECK S1, S5, S15, S16, AND S17

SETF /P4B3T /N4B3T         ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T          ;S5 = 0 1 1 0 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S15 = 1 0 0 0 0
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T          ;S5 = 0 1 1 0 1
CLOCKF CLK
```

```
                                    ;S18 = 1 1 0 0 0
CLOCKF CLK

SETF /P4B3T /N4B3T                  ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T                   ;S5 = 0 1 1 0 1
CLOCKF CLK

SETF P4B3T /N4B3T                   ;S17 = 1 0 0 0 1
CLOCKF CLK

;TEST VECTOR #3 TO CHECK S1, S6, S18, S19, AND S20

SETF /P4B3T /N4B3T                  ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF P4B3T /N4B3T                   ;S6 = 0 1 1 1 0
CLOCKF CLK

SETF /P4B3T /N4B3T                  ;S18 = 1 1 1 0 1
CLOCKF CLK

SETF /P4B3T /N4B3T                  ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF P4B3T /N4B3T                   ;S6 = 0 1 1 1 0
CLOCKF CLK

SETF /P4B3T N4B3T                   ;S19 = 1 1 1 1 0
CLOCKF CLK

SETF /P4B3T /N4B3T                  ;S1 = 0 0 0 1 1
CLOCKF CLK

SETF P4B3T /N4B3T                   ;S6 = 0 1 1 1 0
CLOCKF CLK

SETF P4B3T /N4B3T                   ;S20 = 1 0 1 0 1
CLOCKF CLK

;TEST VECTOR #4 TO CHECK S2, S7, S20, S18, AND S21

SETF /P4B3T N4B3T                   ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T                  ;S7 = 0 0 1 0 0
CLOCKF CLK

SETF /P4B3T /N4B3T                  ;S20 = 1 0 1 0 1
CLOCKF CLK

SETF /P4B3T N4B3T                   ;S2 = 0 0 1 1 1
CLOCKF CLK
```

```
SETF /P4B3T /N4B3T          ;S7 = 0 0 1 0 0
CLOCKF CLK

SETF /P4B3T N4B3T           ;S18 = 1 1 1 0 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T          ;S7 = 0 0 1 0 0
CLOCKF CLK

SETF P4B3T /N4B3T           ;S21 = 1 0 1 1 1
CLOCKF CLK

;TEST VECTOR #5 TO CHECK S0, S2, S7, S8, S22, AND S23

SETF /P4B3T N4B3T           ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T          ;S7 = 0 0 1 0 0
CLOCKF CLK

SETF P4B3T N4B3T            ;S0 = 0 0 0 0 0, ERROR
CLOCKF CLK

SETF /P4B3T N4B3T           ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S8 = 0 0 1 0 1
CLOCKF CLK

SETF /P4B3T /N4B3T          ;S14 = 1 0 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S8 = 0 0 1 0 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S22 = 1 1 0 0 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S8 = 0 0 1 0 1
CLOCKF CLK

SETF P4B3T /N4B3T           ;S23 = 1 0 1 1 0
CLOCKF CLK

SETF /P4B3T N4B3T           ;S2 = 0 0 1 1 1
CLOCKF CLK
```

```
SETF /P4B3T N4B3T          ;S8 = 0 0 1 0 1
CLOCKF CLK

SETF P4B3T N4B3T           ;S0 = 0 0 0 0 0, ERROR
CLOCKF CLK

;TEST VECTOR #6 TO CHECK S2, S9, S24, S25, AND S23

SETF /P4B3T N4B3T          ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF P4B3T /N4B3T          ;S9 = 0 0 1 1 0
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S24 = 1 0 1 0 0
CLOCKF CLK

SETF /P4B3T N4B3T          ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF P4B3T /N4B3T          ;S9 = 0 0 1 1 0
CLOCKF CLK

SETF /P4B3T N4B3T          ;S25 = 1 1 1 0 0
CLOCKF CLK

SETF /P4B3T N4B3T          ;S2 = 0 0 1 1 1
CLOCKF CLK

SETF P4B3T /N4B3T          ;S9 = 0 0 1 1 0
CLOCKF CLK

SETF P4B3T /N4B3T          ;S23 = 1 0 1 1 0
CLOCKF CLK

;TEST VECTOR #7 TO CHECK S3, S10, S16, S26, AND S15

SETF P4B3T /N4B3T          ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S10 = 0 1 0 0 0
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S16 = 1 1 0 0 0
CLOCKF CLK

SETF P4B3T /N4B3T          ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF /P4B3T /N4B3T         ;S10 = 0 1 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T          ;S26 = 1 1 0 1 1
CLOCKF CLK

SETF P4B3T /N4B3T          ;S3 = 0 1 0 1 1
CLOCKF CLK
```

```
SETF /P4B3T /N4B3T          ;S10 = 0 1 0 0 0
CLOCKF CLK

SETF P4B3T /N4B3T           ;S15 = 1 0 0 0 0
CLOCKF CLK

;TEST VECTOR #8 TO CHECK S3, S11, S27, S28, AND S22

SETF P4B3T /N4B3T           ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S11 = 0 1 0 0 1
CLOCKF CLK

SETF /P4B3T /N4B3T          ;S27 = 1 0 0 1 0
CLOCKF CLK

SETF P4B3T /N4B3T           ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S11 = 0 1 0 0 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S28 = 1 1 0 1 0
CLOCKF CLK

SETF P4B3T /N4B3T           ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF /P4B3T N4B3T           ;S11 = 0 1 0 0 1
CLOCKF CLK

SETF P4B3T /N4B3T           ;S22 = 1 1 0 0 1
CLOCKF CLK

;TEST VECTOR #9 TO CHECK S0, S3, S12, S13, S28, AND S25

SETF P4B3T /N4B3T           ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF P4B3T /N4B3T           ;S12 = 0 1 0 1 0
CLOCKF CLK

SETF /P4B3T /N4B3T          ;S13 = 1 1 1 1 1
CLOCKF CLK

SETF P4B3T /N4B3T           ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF P4B3T /N4B3T           ;S12 = 0 1 0 1 0
CLOCKF CLK

SETF /P4B3T N4B3T           ;S28 = 1 1 0 1 0
CLOCKF CLK

SETF P4B3T /N4B3T           ;S3 = 0 1 0 1 1
CLOCKF CLK
```

```
SETF P4B3T /N4B3T              ;S12 = 0 1 0 1 0
CLOCKF CLK

SETF P4B3T /N4B3T              ;S25 = 1 1 1 0 0
CLOCKF CLK

SETF P4B3T /N4B3T              ;S3 = 0 1 0 1 1
CLOCKF CLK

SETF P4B3T N4B3T               ;S0 = 0 0 0 0 0, ERROR
CLOCKF CLK


TRACE_OFF
```

PALC22V10 Creates Manchester Encoder Circuit

AN-167

Manchester Encoding

The PALC22V10 has been programmed to give a Manchester encoded output for eight digital TTL-level inputs or a bit-serial TTL level input. The reason for using such encoding is to remove the DC component from a digital waveform prior to transmission. If the data were transmitted without encoding, 'DC wander' could result.

A transmitted binary signal consists of a series of High and Low voltage pulses. If the signal consists of primarily High pulses, a mean DC component will exist. If the signal then becomes primarily Low pulses, the mean DC component will shift, or wander. This can cause transmission problems on capacitive lines. A Manchester encoded signal cannot have a DC component.

The principle of Manchester encoding is to introduce phase shifting in a carrier wave, which is continuously switching, such that logic input changes are represented by a phase shift of that carrier. If data at the input of an encoder switches from a logic High to a logic Low, then a phase shift of 180 degrees (equivalent to a polarity shift) is performed. When a Low-to-High transition occurs at the input, the carrier phase is switched back by 180 degrees to the original phase.

For example, a logic High could be encoded as a zero phase shift, and a logic Low as a 180-degree phase shift from a reference. The result in the encoded signal is that two successive Highs or Lows represent either a positive or negative data transition (Figure 1). When the data in the figure toggles Low to High the encoded output gives two successive High levels as the phase of the carrier shifts by 180 degrees. The transition of data from High to Low will generate two successive Low outputs in the encoded signal, returning the signal to the original phase.

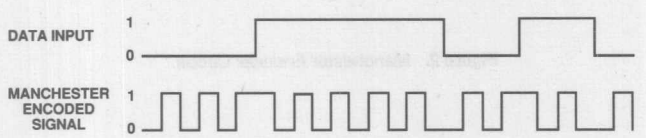


Figure 1. A Manchester Encoded Signal from a Digital TTL Input

Applications

Manchester encoding is applied to local area networks, such as Ethernet, and can be used in storage of data on magnetic media, or in other transmissions including infrared. Both data and clock are imbedded in the Manchester encoded signal. Clock information may easily be extracted from the encoded waveform by using phase-locked loop techniques, and the clock can be used to recover the encoded data. Thus, the data can be resynchronized on reception.

Circuit

Figure 2 shows a schematic diagram of the Manchester encoder circuit. The design has been made to be compatible with either serial or parallel data output from standard UART (Universal Asynchronous Receiver/Transmitter) circuits. The clock input to the system will come from the clock input to the UART circuit, which is typically sixteen times the data rate. The divide-by-sixteen circuit will generate a square wave output at R3, the original data rate as applied to the UART. This signal is available as an output from the system. On a PCB, the R3 signal can be used to decode the data.

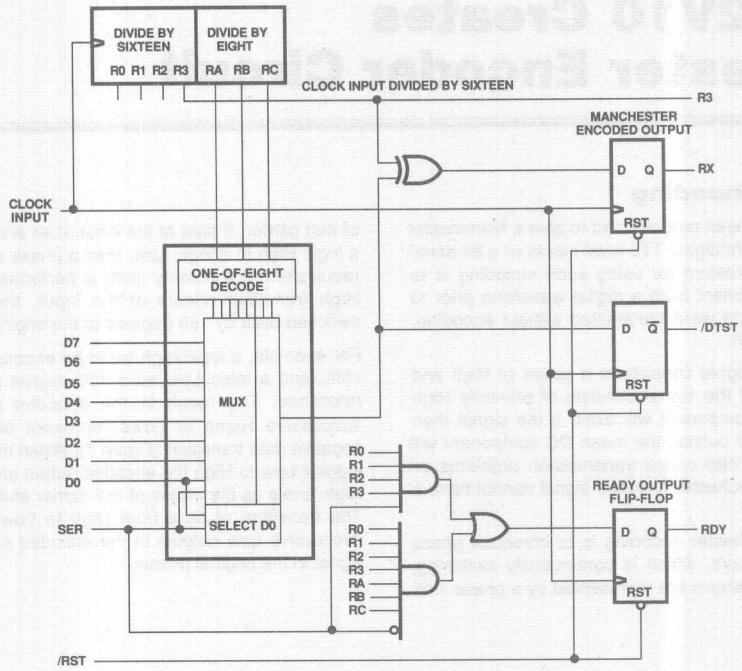


Figure 2. Manchester Encoder Circuit

A further counter synchronously clocked is a divide-by-eight counter with outputs RA, RB and RC. These outputs will select, in the case of a parallel data input, one of the individual data lines D0-D7. The selected data line is routed through the multiplexer and applied to the input of an eXclusive-OR (XOR) gate. One of the inputs to the XOR gate is the data clock rate, available at the R3 flip-flop output. The function of the XOR gate has been configured in the PALC22V10 from a 'sum-of-products' architecture.

The phase encoded signal has been derived at the output of flip-flop RX. Inverting the data gives a 180-degree phase shift from the reference established at the R3 output. The selected data input line determines whether or not phase (polarity) inversion should take place. If the data input is Low, no inversion takes place; if High, the reference signal is inverted to give the required phase shift.

Equations

The general output equation is:

$$\overline{RX} := R3*\overline{DATA} + \overline{R3}*DATA \; (= R3 :+: DATA). \qquad (1)$$

The data input is selected by a one-of-eight condition of the state machine RA, RB, and RC. RA is the clock reference divided by two, RB is the clock divided by four and RC by eight.

For a parallel input, data D0 is selected to control the output polarity at RX when $\overline{RA}*\overline{RB}*\overline{RC}$ is True, D1 when $RA*\overline{RB}*\overline{RC}$ is True, D2 when $\overline{RA}*RB*\overline{RC}$ is True, etc. This procedure serializes the parallel data. The Manchester encoded signal is created by this dynamic control of the inverting input to the XOR configuration through the selected data input.

There is a serial select input to the multiplexer circuit such that when driven active High, serial data applied to the D0 input will encode the carrier signal on a bit-by-bit basis. Input lines D1-D7 are ignored when the encoder is used in this mode. The SER mode should be hard-wired Low for parallel data input.

To enable external handshake to be accomplished in either serial or parallel mode, a RDY output is available. When a complete byte or bit is encoded and transmitted to the output RX, the RDY pin will go High to request more data. Note that the Manchester encoder does not latch the data; it must remain valid for the duration of the encoding. The logic circuit required for the RDY pin is shown as a sum of product terms as applied to the 'D' input of the RDY flip-flop.

The data that is applied to the input is available in a serial form at the DTST output and can be used for test purposes. DTST is simply the serialized output of the data input D0-D7.

PAL Device Design Considerations

The PALC22V10 was chosen for this application because of the large number of product terms required to encode the XOR function. Up to sixteen product terms can be used to create an XOR function on eight data inputs. The PALC22V10 also offers low power with TTL compatibility, reducing power supply requirements while increasing reliability.

The flip-flops R0, R1, R2, R3, RA, RB, and RC form a long binary count chain. The output from R3 forms the reference phase output which is the clock input divided by sixteen. As discussed, RA, RB, and RC form a binary state machine to select D0-D7 data inputs individually.

Design Procedure

The following figures show the design procedure used to derive equations for the binary divide-by-sixteen counter R0-R3. The Boolean equations have been derived using the PALASM® 2 software operators.

Figure 3 shows the counter states in hexadecimal and binary formats. Figure 4 shows two Karnaugh maps, the state assignment and state transition; Figure 4a shows the hexadecimal value of the current state of the counter, and Figure 4b shows the next state. For example, the location in the state assignment map of hex 'A' would be represented by the next state hex 'B' in the transition map.

State	R0	R1	R2	R3
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	1	0	1	0
6	0	1	1	0
7	1	1	1	0
8	0	0	0	1
9	1	0	0	1
A	0	1	0	1
B	1	1	0	1
C	0	0	1	1
D	1	0	1	1
E	0	1	1	1
F	1	1	1	1

Figure 3. Binary Counter Truth Table. The Conditions of the Flip-flops R0, R1, R2, and R3 Are Shown in the Truth Table with the Hexadecimal Value of the Binary Weighting

R3 R2 \ R1 R0	10	11	01	00
10	A	B	9	8
11	E	F	D	C
01	6	7	5	4
00	2	3	1	0

4a. State Assignment Map

R3 R2 \ R1 R0	10	11	01	00
10	B	C	A	9
11	F	0	E	D
01	7	8	6	5
00	3	4	2	1

4b. State Transition Map

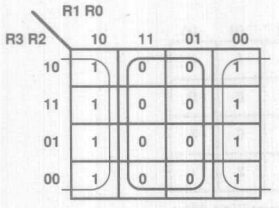
Figure 4. The State Assignment Map (4a) Shows the Current Hexadecimal Value in the Appropriate Position in the Map. The State Transition Map (4b) Shows Next State. For Example, State A in the Assignment Map is Shown as B in the Transition Map Because that is the Next Count in the Sequence

Figure 5 shows the binary weighting of the hexadecimal values of the flip-flops R0, R1, R2, and R3 of the transition map. From this map, individual Karnaugh maps are drawn for each individual flip-flop, and equations derived by performing minimization for each flip-flop.

R3 R2 \ R1 R0	10	11	01	00
10	1101	0011	0101	1001
11	1111	0000	0111	1011
01	1110	0001	0110	1010
00	1100	0010	0100	1000

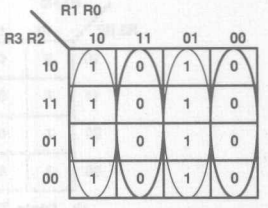
Figure 5. State Transition Map with the Binary Weighting of the Hexadecimal Entries. The Register Significance of the Entry is from Left to Right. For Example, 1101 Represents the Binary Weighting of R0 R1 /R2 R3

Figure 6 shows the Karnaugh maps and the equations for either true or complement outputs. For true outputs, minimization is performed on the logic One entries in the map. Where an inverting buffer is used between the internal flip-flop and the outside world, minimization is performed on the logic Zero entries in the map (shown bolder). This is because the internal flip-flop's output is set High, but the state of the output pin is a logic Low due to the inverting action of the buffer. With the output configuration of the PALC22V10, it is possible to choose either inverted or non-inverted outputs, so either equation will be suitable.



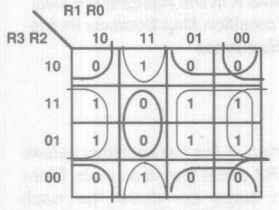
/R0 := R0 (1)
R0 := /R0 (2)

6a. Map for Flip-flop R0.



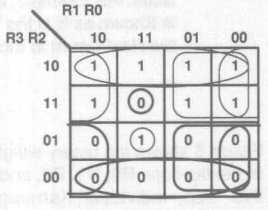
/R1 := R0*R1 + /R0*/R1 (3)
R1 := /R0*R1 + R0*/R1 (4)

6b. Map for Flip-flop R1.



/R2 := R2*R1*R0 + /R2*/R1 + /R2*/R0 (5)
R2 := /R2*R1*R0 + R2*/R1 + R2*/R0 (6)

6c. Map for Flip-flop R2.



/R3 := R3*R2*R1*R0 + /R3*/R2
+ /R3*/R1 + /R3*/R0 (7)
R3 := /R3*R2*R1*R0 + R3*/R2
+ R3*/R1 + R3*/R0 (8)

6d. Map for Flip-flop R3.

Figure 6. The Karnaugh Map of Figure 5. Broken Down for Individual Flip-flops R0, R1, R2, and R3. Minimization is Performed for Each Individual Flip-flop

The PALASM 2 software operators describing the equation are:

- * logical AND
- + logical OR
- / logical inversion
- := registered output equation.

The equations derived for a general counter of length N become:

$$\overline{QN} := QN*QN\text{-}1*QN\text{-}2*...Q2*Q1*Q0 + \overline{QN}*\overline{QN\text{-}1} + \overline{QN}*\overline{QN\text{-}2} : : + \overline{QN}*\overline{Q1} + \overline{QN}*\overline{Q0}$$

using the inverting option, or:

$$QN := \overline{QN}*QN\text{-}1*QN\text{-}2*...Q2*Q1*Q0 + QN*\overline{QN\text{-}1} + QN*\overline{QN\text{-}2} : : + QN*\overline{Q1} + QN*\overline{Q0}$$

for the non-inverting option.

This generalized equation requires N + 1 product terms for a general flip-flop, QN. For example, Q7, in a binary count sequence, will require a summation of eight product terms. The complete design specification is shown in the PAL device Design Specification of Pattern 04.

Design Results

Figure 7 is an oscilloscope photograph of the outputs of the programmed device. The top trace is the RDY signal. It goes High to indicate that a complete data byte has been transmitted and that the next data byte may be presented to the data inputs D0-D7. The second trace is $\overline{DTST}$, the serial data stream D0-D7 of the parallel data input D0-D7. The bottom trace is the Manchester encoded waveform of the parallel data input, or RX. The markers show one complete frame of information.

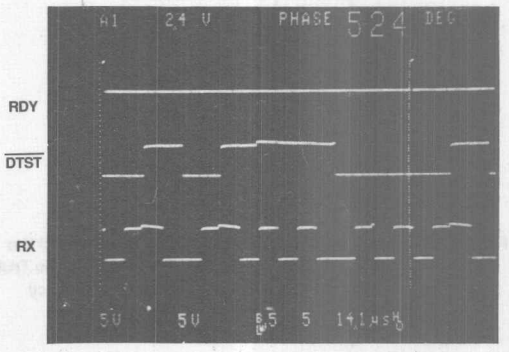


Figure 7. Oscilloscope Trace of Programmed Device. The Manchester Encoded Output is RX

```
TITLE           MANCHESTER ENCODER.
PATTERN         04.
REVISION        01.
AUTHOR          CHRIS JAY.
COMPANY         MMI SANTA CLARA, CA.
DATE            17 JULY 1986.
;
;THE PALC22V10 HAS BEEN DESIGNED TO FUNCTION AS A MANCHESTER
;ENCODER CIRCUIT.  THE BINARY DATA INPUT WILL DETERMINE WHETHER
;THE TRANSMISSION SIGNAL SHOULD BE IN PHASE WITH A CLOCK
;REFERENCE OR SHIFTED BY 180 DEGREES FROM THAT REFERENCE.
;THE CHARACTERISTIC OF A MANCHESTER ENCODED SIGNAL IS A
;CHANGE OF PHASE FOR EACH TRANSITION OF THE DATA INPUT.
;ASSUMING THAT 'X' IS THE PHASE OF THE CARRIER THAT ENCODES
;A LOGIC LOW INPUT, WHEN THE INPUT DATA TRANSITIONS
;TO A LOGIC HIGH THE PHASE OF THE CARRIER WILL BE SHIFTED
;TO 'X + 180'.  IT WILL CHANGE BACK TO 'X' ONLY WHEN THE
;DATA INPUT TOGGLES FROM A LOGIC HIGH TO A LOGIC LOW.
;THE KEY FEATURES OF THE ENCODED SIGNAL ARE:
;
;       1. DATA AND CLOCK FREQUENCY INFORMATION IS
;          CONTAINED IN THE ENCODED SIGNAL.  CLOCK
;          INFORMATION MAY BE EXTRACTED BY USING A
;          PHASE LOCKED LOOP IN THE RECEIVER.
;       2. NO DC COMPONENT IS CONTAINED IN THE
;          PHASE ENCODED SIGNAL.
;
;THE APPLICATIONS OF PHASE ENCODED SIGNALS CAN BE IN SHORT
;COMMUNICATIONS LINKS, SUCH AS ETHERNET OR INFRARED TRANS-
;MISSIONS, OR IN STORAGE ON SOME MAGNETIC MEDIA.
;
;THERE ARE EIGHT DATA INPUTS, D0 TO D7.  DATA PRESENT ON
;THESE PINS WILL BE ENCODED INTO A BI-PHASE SIGNAL AT THE
;OUTPUT QX, WHEN THE SER (SERIAL) INPUT DATA PIN IS CONFIGURED
;TO A LOGIC ZERO.  TO CONVERT A SERIAL DATA STREAM OF LOGIC
;ONES AND ZEROS INTO A PHASE ENCODED SIGNAL THE SER INPUT
;IS TIED HIGH, AND THE DATA IS APPLIED TO THE D0 INPUT
;OF THE PAL DEVICE.  A CLOCK INPUT OF 16 TIMES THE DATA RATE
;IS REQUIRED FOR THE PARALLEL OR SERIAL ENCODING OF THAT
;DATA, SO THE DESIGN MAY BE USED WITH MANY POPULAR UART LSI
;CIRCUITS.  THERE IS ONE HANDSHAKE OUTPUT FROM THE DEVICE,
;RDY.  THIS SIGNAL GOES HIGH WHEN THE NEXT DATA BYTE OR BIT
;IS REQUIRED TO BE SET UP AT THE INPUT.
```

```
CHIP MANENC PAL22V10
;
;PIN     1       2       3       4       5       6
        CLK     /RST    D0      D1      D2      D3

;PIN     7       8       9       10      11      12
        D4      D5      D6      D7      NC      GND

;PIN     13      14      15      16      17      18
        SER     RDY     /DTST   RC      RB      RX

;PIN     19      20      21      22      23      24
        RA      R3      R2      R1      R0      VCC

GLOBAL                                  ;GLOBAL TERM
                                        ;ENABLES RESET.
                                        ;STRING DECLARATIONS.
STRING S0 '/RC*/RB*/RA'                 ;ENCODE STATE 0
STRING S1 '/RC*/RB* RA'                 ;ENCODE STATE 1
STRING S2 '/RC* RB*/RA'                 ;ENCODE STATE 2
STRING S3 '/RC* RB* RA'                 ;ENCODE STATE 3
STRING S4 ' RC*/RB*/RA'                 ;ENCODE STATE 4
STRING S5 ' RC*/RB* RA'                 ;ENCODE STATE 5
STRING S6 ' RC* RB*/RA'                 ;ENCODE STATE 6
STRING S7 ' RC* RB* RA'                 ;ENCODE STATE 7
                                        ;FOR PARALLEL INPUT
                                        ;S0 SELECTS DO INPUT,
                                        ;S1 SELECTS D1 INPUT,
                                        ;ETC.
EQUATIONS                               ;
                                        ;
GLOBAL.SETF = RST                       ;SET INITIAL
                                        ;CONDITIONS, RESET.
/R0   :=  R0                            ;R0 DIVIDES THE CLOCK
                                        ;INPUT BY 2.
/R1   :=  R1* R0                        ;R1 DIVIDES THE CLOCK
      + /R1*/R0                         ;BY 4.
                                        ;
/R2   :=  R2* R1* R0                    ;R2 DIVIDES THE CLOCK
      + /R2*/R1                         ;BY 8.
      + /R2*    /R0                     ;
                                        ;
/R3   :=  R3* R2* R1* R0                ;R3 DIVIDES THE CLOCK
      + /R3*/R2                         ;BY 16.
      + /R3*    /R1                     ;
      + /R3*       /R0                  ;
                                        ;RDY OUTPUT PERFORMS
RDY   := RC*RB*RA*R3*R2*R1*R0*/SER      ;A HANDSHAKE OPERATION.
      +          R3*R2*R1*R0* SER       ;WHEN HIGH, NEW DATA
                                        ;MAY BE APPLIED.
```

```
/RA   :=  RA* R3* R2* R1* R0*/SER          ;COUNTER [RA, RB, RC]
      + /RA*/R3*          /SER             ;COUNTS THE NUMBER
      + /RA*    /R2*      /SER             ;OF BITS TRANSMITTED
      + /RA*       /R1*   /SER             ;WHEN A BYTE IS
      + /RA*          /R0*/SER             ;APPLIED TO THE
      +                SER                 ;D0 - D7 DATA INPUTS.
                                           ;THE LOGIC CONDITION
/RB   :=  RB* RA* R3* R2* R1* R0*/SER      ;OF THE SELECTED
      + /RB*    /R3*          /SER         ;INPUT IS MULTIPLEXED
      + /RB*       /R2*       /SER         ;TO THE RX REGISTERED
      + /RB*          /R1*    /SER         ;OUTPUT BY THE 'ONE OF
      + /RB*             /R0*/SER         ;EIGHT' STATE SELECTOR,
      + /RB*/RA*            /SER          ;IN THE DIVIDE BY EIGHT
      +                      SER           ;COUNTER RA, RB, RC.  FOR
                                           ;SERIAL MODE /RA*/RB*/RC
/RC   :=  RC* RB* RA* R3* R2* R1* R0*/SER  ;SELECTS DATA INPUT D0,
      + /RC*       /R3*          /SER      ;RA*/RB*/RC SELECTS D1,
      + /RC*          /R2*       /SER      ;ETC., WHILE /SER SELECTS
      + /RC*             /R1*    /SER      ;PARALLEL OPERATION.  IF
      + /RC*                /R0*/SER      ;SER IS ACTIVE, SERIAL
      + /RC*    /RA*           /SER       ;ENCODING IS SELECTED.
      + /RC*/RB*               /SER       ;THE SERIAL DATA APPLIED
      +                         SER       ;TO THE D0 INPUT IS
                                           ;ENCODED.  D1 - D7
                                           ;INPUTS ARE DESELECTED.
/DTST:=  S0*D0                             ;D0 - D7 IS SERIALIZED
      +  S1*D1*/SER                        ;FOR PARALLEL MODE.
      +  S2*D2*/SER                        ;FOR SERIAL MODE, D0 ONLY
      +  S3*D3*/SER                        ;IS CLOCKED THROUGH TO
      +  S4*D4*/SER                        ;THE DTST OUTPUT.
      +  S5*D5*/SER                        ;/DTST OUTPUT IS THE
      +  S6*D6*/SER                        ;SERIALIZED ENCODING
      +  S7*D7*/SER                        ;OF THE DATA INPUT
                                           ;D0 - D7, OR D0.
/RX   :=  S0*/R3*D0       + S0*R3*/D0     ;THE RX OUTPUT PROVIDES
      +  S1*/R3*D1*/SER + S1*R3*/D1*/SER   ;THE MANCHESTER ENCODED
      +  S2*/R3*D2*/SER + S2*R3*/D2*/SER   ;OUTPUT OF THE REFERENCE
      +  S3*/R3*D3*/SER + S3*R3*/D3*/SER   ;FREQUENCY, WHICH OCCURS
      +  S4*/R3*D4*/SER + S4*R3*/D4*/SER   ;AT THE R3 OUTPUT.  THE
      +  S5*/R3*D5*/SER + S5*R3*/D5*/SER   ;PHASE AT R3 IS SHIFTED
      +  S6*/R3*D6*/SER + S6*R3*/D6*/SER   ;AT THE RX OUTPUT IF THE
      +  S7*/R3*D7*/SER + S7*R3*/D7*/SER   ;SELECTED DATA INPUT PO-
                                           ;LARITY CHANGES.  IF /SER
                                           ;IS INACTIVE, D0 - D7
                                           ;INPUTS ARE SELECTED.  IF
                                           ;/SER IS ACTIVE, ONLY THE
                                           ;SERIAL INPUT APPLIED TO
                                           ;D0 IS SELECTED.
```

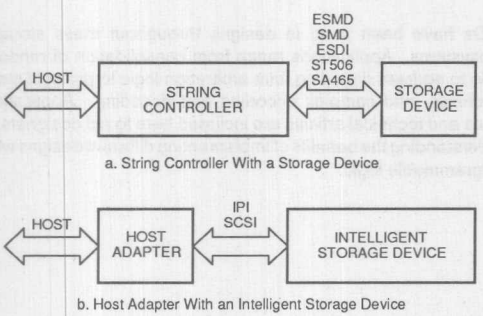
```
SIMULATION                                ;
TRACE_ON CLK R0 R1 R2 R3 RA RB RC RX      ;TRACE ALL SIGNALS.
         /DTST RDY D0 D1 D2 D3 D4 D5      ;
         D6 D7 SER /RST                   ;SET INITIAL CONDITIONS
SETF     /CLK RST D0 D1 /D2 D3 D4 /D5     ;FOR DATA & CONTROL IN-
         /D6 /D7 /SER                     ;PUTS. PARALLEL MODE SET.
SETF     RST                              ;RESET ACTIVE TO
CLOCKF   CLK                              ;INITIALIZE THE
SETF     /RST                             ;REGISTERS
FOR I := 1 TO 128 DO                      ;CLOCK 8 DATA BITS
BEGIN CLOCKF CLK                          ;THROUGH TO R3 & RX
END                                       ;OUTPUTS.
SETF SER D0                               ;SET SERIAL MODE VIA
FOR I := 1 TO 32 DO                       ;SERIAL CONTROL PIN.
BEGIN CLOCKF CLK                          ;CLOCK SYSTEM TO SELECT
END                                       ;SERIAL OPERATION.
TRACE_OFF                                 ;
```

Peripherals are integral to any computer system. They range from basic keyboards and printers to mass storage disks/tapes. In this section we will focus on mass storage systems such as disk drives and tape drives.

A typical mass storage subsystem can be thought of in terms of two major sections: the controller and the storage device. The controller provides an interface between the host and the storage device. It may control one or several storage devices at a time. Common storage devices are disk drives (winchester and floppy) and tape drives (reel and cartridge).

The discussion that follows will be oriented toward the storage device being a disk drive although it could just as easily have been done using tape drives.

Figure 1 shows two configurations of disk storage subsystems. In Figure 1A the storage device is connected to the host via a string controller. Communication between the storage device and the controller is accomplished through standardized interface protocols. Examples of common disk drive interfaces in use today are SMD (Storage Module Device), ESMD (Enhanced SMD), ESDI (Enhanced Small Disk Interface), ST506, and SA465. An application example of an ESDI interface controller is shown on page 2-509. An intelligent storage device is created when the string controller function is embedded into the storage device (Figure 1B). Intelligent storage devices typically communicate to the host through a host adapter using either SCSI (Small Computer System Interface) or IPI (Intelligent Peripheral Interface) interface protocols.

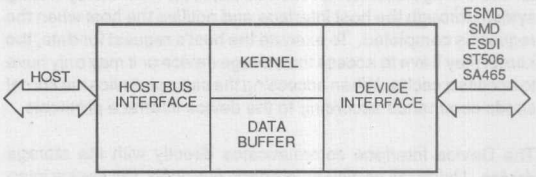


412 01

Figure 1.

Controller

The controller function can be conceptually divided into several sections: the Host Bus Interface, the Data Buffer, the Kernel and the Device Interface.



412 02

Figure 2.

The Host Bus Interface provides the connections to the host CPU and main memory. It can serve as either a slave or a master on the host bus. When the controller is a bus slave, the host CPU must be able to access the controller, pass commands, and obtain subsystem status using established host bus protocol. The controller behaves as a bus master while transferring data in either direction, or posting status. A bus master must be able to arbitrate for the host bus and execute the necessary handshake protocols.

The Data Buffer stores the data being read from or written to the disk and checks for data integrity. The physical buffer can serve several purposes: data transfer rate adaptor between the host and the disk, a temporary storage for data going to the host until it can be verified error free, and a cache. Depending on the complexity of the data buffer it may be as small as one byte or as large as a complete track of data.

Data integrity can be monitored for data transfers from the host as well as from the disk drive. The sophistication used to monitor data integrity varies significantly from simple single bit error detection (parity) to large multiple bit error detection and correction. The host may elect to transfer data with a parity bit for each byte. Data error detection and correction schemes are employed between the controller and the disk drive. As storage densities have increased, so have the size of the errors. The demand for more capable error detection and correction codes that can more accurately detect and correct larger number of bits in error has increased accordingly.

Such error detection codes as CRC code are satisfactory if the error occurs in the data transmission channel and the data can be recovered by retransmission. Data errors caused by media defects require the capability of both error detection and correction. As the nature of media errors has changed, so have the error detection and correction codes being used. Fire Codes and computer generated codes have been the most popular codes. Symbol oriented codes, such as Reed-Solomon codes, are emerging as the solutions for the future.

The Kernel is the brain of the controller and the manager of the subsystem. It translates commands from the host into commands for the storage device. It receives requests from the operating system through the host interface and notifies the host when the request is completed. To execute the host's request for data, the kernel may have to access the storage device or it may only have to access a cache. When accessing the storage device the kernel sends commands according to the device interface protocols.

The Device Interface communicates directly with the storage device. Using the particular interface protocols, the device interface sends commands, retrieves status, and transfers data between the controller and the storage device. The device interfaces shown in Figure 3 are predominantly disk and would have to be changed to accommodate tape drives. An application example of a tape drive interface controller is shown on page 2-519.

Storage Device

The storage device can also be divided into several sections: the Interface & Control, the Servo and the Read/Write.

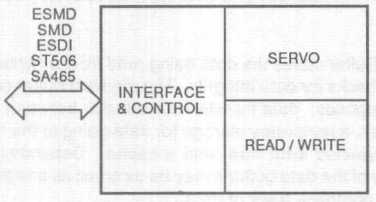


Figure 3.

The Interface & Control is the brain of the storage device. It interprets the commands from the controller. It supervises the servo and the read/write and maintains the device status.

The Servo is responsible for the movement of the media and the positioning of the read/write heads. For disk drives D.C. brushless motors are predominantly used to drive the spindle. Head positioning is done either with a stepper motor or a voice coil actuator. In the case of tape drives, the Servo is only responsible for capstan motor control because the position of the read/write heads is fixed.

The Read/Write is responsible for storing and retrieving data from the media. The data encoding method must balance data density and ease of data recovery. Several Run-Length-Limited (RLL) coding techniques are in use today.. Winchester disk drives have used frequency modulation (FM), modified FM (MFM), modified MFM (MMFM), and are currently using 2,7 to obtain present day densities. Tape drives employ different RLL codes than disk drives. The two common codes in use for tape storage are phase encoding (PE) and group-coded recording (GCR). The application note on page 2-541 shows an example of a GCR encoder implemented with PAL devices.

When the storage device is configured with an embedded controller (Figure 4) the *Device Interface* section of the controller and the *Interface & Control* section of the storage device are not required. The Kernel will directly control the storage device.

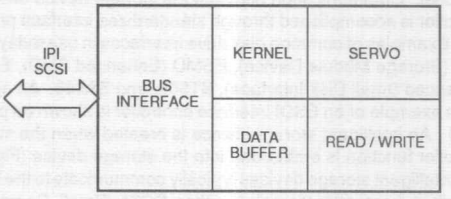


Figure 4.

PLDs have been used in designs throughout mass storage subsystems. Applications range from consolidation of random logic to address decoding, bus arbitration logic to protocol state machines, and complex encoding and decoding. Application notes and technical articles are included here to aid designers in understanding the benefits of implementing different designs with programmable logic.

Building an ESDI Translator Using the M2064 Logic Cell Array

AN-179

The ESDI Translator

ESDI is a low-cost, high-performance interface standard suitable for the smaller, high-performance Winchester disk drives currently being produced. The ESDI interface consists of a control cable and a data cable. The control cable allows for a daisy chain connection of up to seven devices (disk or optical drives) with only the last device being terminated. In our design, we assumed that the device is a disk drive. The data cable is attached in a radial configuration (See Figure 1).

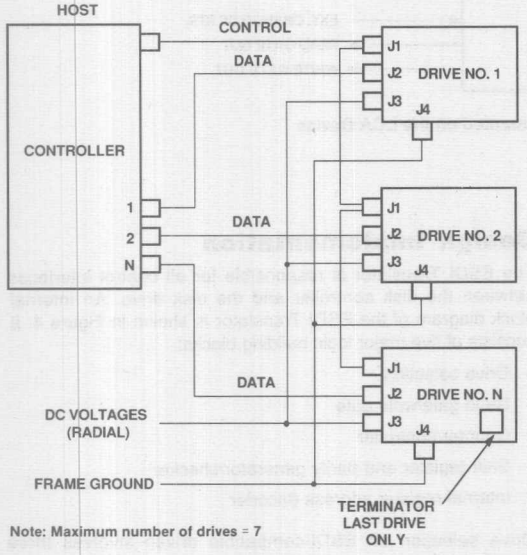


Figure 1. Connection Between the Controller and Multiple Drives

The ESDI Translator handshakes serial commands from a disk controller, deserializes the commands and passes the commands to a microcontroller. The command data word structure is shown in Figure 2.

The Command Function bits define functions to be executed by the disk drive. These functions are seek, recalibrate, request status, request configuration, select head group, control, data strobe offset, track offset, initiate diagnostics, set bytes per sector and set configuration. Some of these functions, such as the control function, have modifiers for more detailed functional description. Other commands have parameters that contain numbers. For the seek command, the parameter specifies the cylinder number that the drive will seek to. The request status and request configuration commands require data from the disk drive to be transferred back to the disk controller. In our current design, internal registers A and B in the LCA device represent the upper and lower bytes of the command respectively.

Figure 3 illustrates the relationships between the disk controller, the ESDI Translator, and the microcontroller. The PROM is used to store the configuration data for the LCA device. The Done/Program (D/$\overline{P}$) output is driven LOW when the device is being configured. Configuration data is read from the PROM device during configuration. After configuration is complete, the LCA device drives the D/P pin HIGH to deselect the PROM. Drive selection, write protection, command completion and fault detection are also handled by the ESDI Translator.

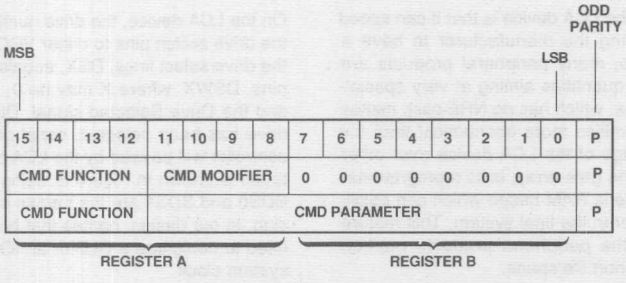


Figure 2. Command Data Word Structure

Logic Cell™ Array and XACT™ are trademarks of XILINX, Inc.
P-SILOS™ is a trademark of SimuCad Corp.

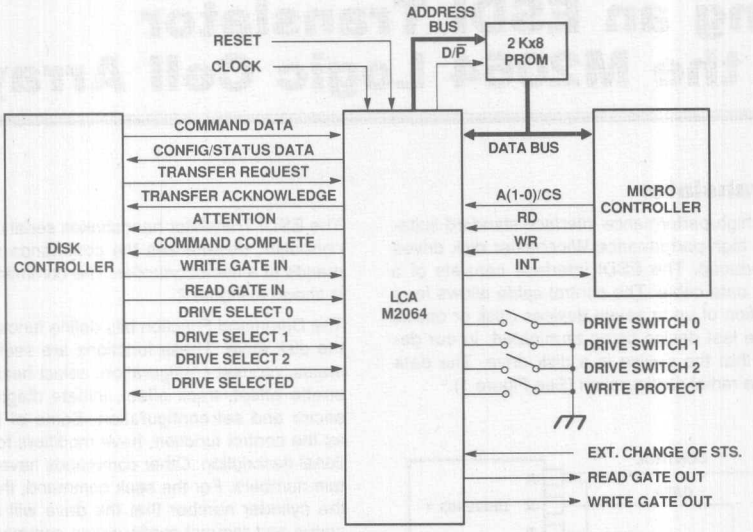


Figure 3. An ESDI Translator Implemented on the LCA Device

Why Use an LCA Device in an ESDI Translator

The ESDI interface standard requires more logic functions to be built into the disk drive than some other interface standards, such as the ST506 standard. However, the external dimensions of a disk drive usually have to conform to an industrial standard form factor. Thus, the use of high-density semicustom chips is the logical solution to increase functionality without increasing the external dimensions.

The LCA device is a high-density CMOS integrated circuit available from Monolithic Memories. Its high gate density allows the implementation of an ESDI Translator in a single chip. Fifteen standard SSI and MSI chips would be necessary for the same application. If PLDs are used to implement the ESDI Translator, more than one would be necessary because a large amount of logic is required, thus occupying more board space.

A major advantage in using the LCA device is that it can speed up the design cycle, enabling the manufacturer to have a shorter time-to-market. Also, many peripheral products are produced in relatively small quantities aiming at very specialized markets. The LCA device, which has no NRE cost, makes the production of small quantities more economical than the gate array. Another advantage of the LCA device over other semicustom chips, such as the gate array, is its reprogrammability feature. The LCA device is RAM-based which can easily be reprogrammed by the user in the final system. This feature is especially important in the peripheral products market, where many products have short life spans.

Design Implementation

The ESDI Translator is responsible for all control interfaces between the disk controller and the disk drive. An internal block diagram of the ESDI Translator is shown in Figure 4. It consists of five major logic building blocks:

- Drive selection
- Read gate/write gate
- Counter/controller
- Shift register and parity generator/checker
- Internal register address decoder

Drive selection on ESDI-compatible drives involves three signals from the disk controller, Drive Select 0-2. These three drive select lines are encoded so that up to seven drives may be connected to the same ESDI port, as shown in Table 1.

On the LCA device, the drive number is selected by connecting the drive switch pins to either VCC or GND. When the code on the drive select lines, DSX, equals the code on the drive switch pins, DSWX, where X may be 0, 1, or 2, the drive is selected and the Drive Selected signal, DSELD, is asserted. Once the drive has been selected, serial commands output by the disk controller will be read by the LCA device. The actual implementation is shown in Figure 5, using two CLBs and seven IOBs. BDS0 and BDS1 are the names of the CLBs in the current design. In our design, names that begin with the letter B or P are used to designate a CLB or an IOB respectively. SCLK is the system clock.

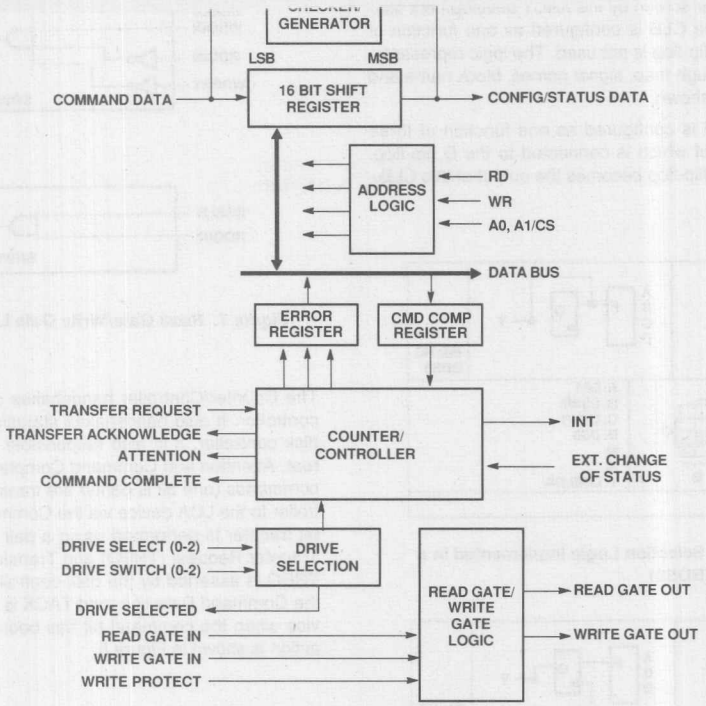


Figure 4. ESDI Translator Internal Block Diagram

DS2/ DSW2	DS1/ DSW1	DS0/ DSW0	DRIVE
0	0	0	None
0	0	1	Select Drive 1
0	1	0	Select Drive 2
0	1	1	Select Drive 3
1	0	0	Select Drive 4
1	0	1	Select Drive 5
1	1	0	Select Drive 6
1	1	1	Select Drive 7

Table 1. Drive Selection

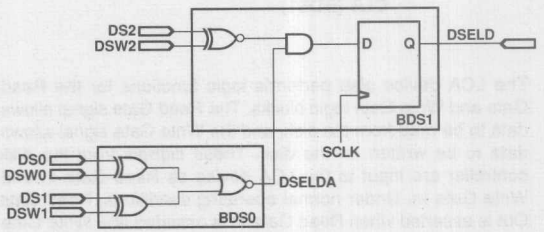


Figure 5. Configuration of the BDS0 and BDS1 CLBs to Provide the Drive Selected Signal

Figure 6 shows the configurations of the two CLBs as displayed on the computer screen by the XACT development software. In Figure 6a, the CLB is configured as one function of four variables. The D flip-flop is not used. The logic representation, truth table, Karnaugh map, signal names, block name and Boolean equation are shown.

In Figure 6b, the CLB is configured as one function of three variables, the output of which is connected to the D flip-flop. The Q output of the D flip-flop becomes the output of this CLB.

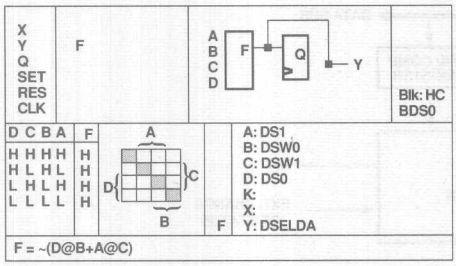


Figure 6a. Drive Selection Logic Implemented in a CLB (BDS0)

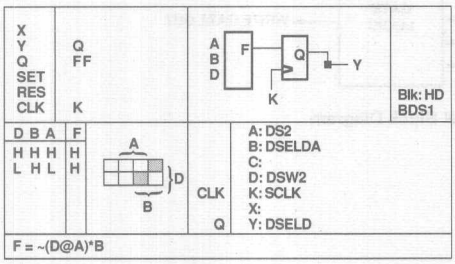


Figure 6b. Drive Selection Logic Implemented in a CLB (BDS1)

The LCA device also performs logic functions for the Read Gate and Write Gate logic blocks. The Read Gate signal allows data to be read from the disk, and the Write Gate signal allows data to be written on the disk. These signals from the disk controller are input to the LCA device as Read Gate In and Write Gate In. Under normal operating conditions, Read Gate Out is asserted when Read Gate In is asserted and Write Gate Out is asserted when Write Gate In is asserted. When both Read Gate In and Write Gate In are asserted, a write error condition results and the Attention line is asserted, signalling the disk controller that an error has occurred. Also, the LCA device may be used to provide write protection to a disk drive. When the Write Protect signal is asserted, the Write Gate Out signal will not be asserted when the Write Gate In signal is asserted, preventing the write circuitry from being activated. These logic functions are implemented in two CLBs, BRWG0 and BRWG1, as shown in Figure 7.

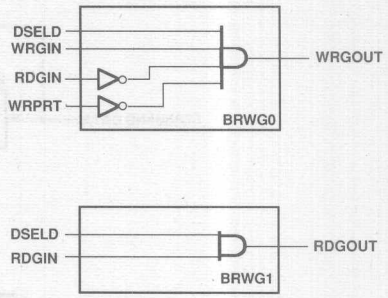


Figure 7. Read Gate/Write Gate Logic Implementation

The Counter/Controller handshakes commands from the disk controller. It also handshakes status/configuration data to the disk controller. It is also responsible for generating the Interrupt, Attention and Command Complete signals. Seventeen bit commands (one bit is parity) are transferred from the disk controller to the LCA device via the Command Data line. The serial bit transfer is performed using a pair of handshaking signals, Transfer Request (TREQ) and Transfer Acknowledge (TACK). TREQ is asserted by the disk controller when a bit is valid on the Command Data line, and TACK is asserted by the LCA device when the command bit has been read. The handshaking action is shown in Figure 8.

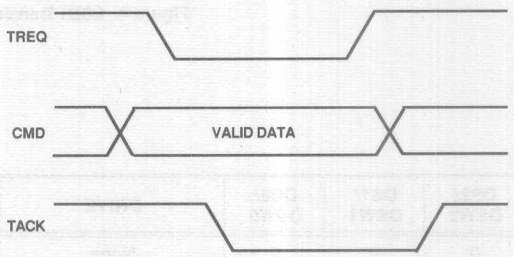


Figure 8. Command Data Transfer

A 17-state counter counts the number of command bits shifted in or shifted out of the data registers. Its implementation is shown in Figure 9. The SHIFT signal is asserted, thus incrementing the counter, whenever there is a transfer request and the LCA device is selected. CNT_Q0 is the lowest bit of the shift register counter. This bit is inverted whenever SHIFT is asserted unless sixteen bits have already been shifted into the LCA device (CNT16 asserted, or CKCMD asserted when all seventeen bits have been shifted in). The DATAOUT signal is asserted after a Request Status command or a Request Configuration command has been transferred and the internal data registers have been loaded with data to be serially shifted out. Hence, when DATAOUT is asserted, CKCMD is negated and the counter is enabled.

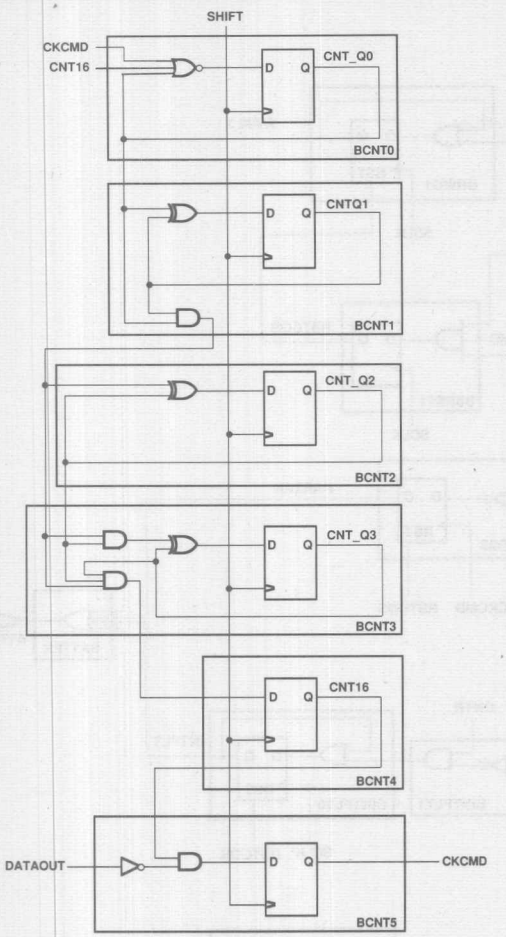


Figure 9. 17-State Counter Implemented in six CLBs

The logic generation of the Attention signal, ATTEN, is shown in Figure 10. Whenever there is a write fault, WRFLT, a parity error, PARERR, an interface fault, INTFLT, or an external error, CSTS(Change of Status), the ATTEN signal is asserted. When both WRGIN and RDGIN signals are asserted and the LCA device is selected, the WRFLT signal is asserted. The WRFLT signal is negated when the command transferred to the LCA device Register A is the Reset command defined by the ESDI standard, which has a Command Function of 0101 (Control) and a Command Modifier of 0000 (Reset Attention and Standard Status). The Command Function and Command Modifier formats are shown in Figure 2.

Three CLBs, BPG0, BPG1, and BPG2, are responsible for generating the PARERR signal. BPG0 is a multiplexer that selects the source of the input to the parity generator/checker (BPG1). If data is being shifted into the LCA device (parity checker mode), the CMDBITA input is chosen. If data is being shifted out of the LCA device (parity generator mode), the RA_Q7 output is chosen. These two signals are also shown in Figure 11. BPG1 is an odd parity generator/checker. In the parity checker mode, whenever an odd number of ones are passed through this CLB, the output is one. This output signal is connected to BPG2, which inverts the signal and asserts or negates the PARERR signal accordingly. If parity is correct, an interrupt is asserted to the microcontroller informing the microcontroller that a command has been received and is ready to be read. In the parity generator mode, the output of BPG1 is the parity bit. BPG1 is clocked by the SHIFT input, which is asserted whenever there is a transfer request and the LCA device is selected. BPG1 is reset by the Reset Parity Generator input, RSTPGEN. This signal is asserted when either INT is asserted or an interface fault is detected.

The Interface Fault signal is asserted when the LCA device is selected and CNTR is negated before seventeen bits have been transferred. CNTR is asserted after the first command bit is shifted in and is negated after the seventeenth bit is shifted out. BINTFLT0 is clocked by the system clock, SCLK, and reset by the RSTCOS signal. The IOB PCSTS is configured as a buffered input to signal external error conditions.

The microcontroller is able to address four register locations in the LCA: two data registers, BRGA and BRGB, one error register, BDMX, and one command complete register, BRC7. Only three bits in the error register are used. They are bit 0 for parity error, bit 4 for interface fault and bit 7 for write fault. Only one bit in the command complete register is used. This is bit 0, which has a value of zero when the command is completed. The addresses and contents of the registers are shown in Table 2.

A1	A0	REGISTER	BIT NUMBER 7 6 5 4 3 2 1 0
0	0	REGISTER A	
0	1	REGISTER B	
1	0	ERROR REGISTER	
1	1	COMMAND COMPLETE REGISTER	

Table 2. Register Addresses and Contents

The LCA device implementation of the registers and multiplexer is shown in Figure 11. When the interrupt signal is asserted, the microcontroller reads the two data registers by setting the A1 and A0 address lines appropriately and asserting the RD signal to the LCA device. These data registers contain the command that is transferred from the disk controller through the CMDBITA input. If the command is a request data command, configuration or status data is written to these two data registers by the microcontroller. These two bytes, plus a parity bit that is generated by the parity generator in the LCA device, are serially transferred to the disk controller over the Config/Status Data line through RA_Q7. After all seventeen bits have been transferred, the Command Complete signal is asserted. If the command is not a request data command, the microcontroller executes the command and upon completion, writes a byte of zeros to the command complete register of the LCA. When the command complete register is written with all zeros, the Command Complete signal is asserted by the LCA device.The command complete register may only be written, not read. The status register contains error bits that are set when errors are detected. This register may only be read and not written.

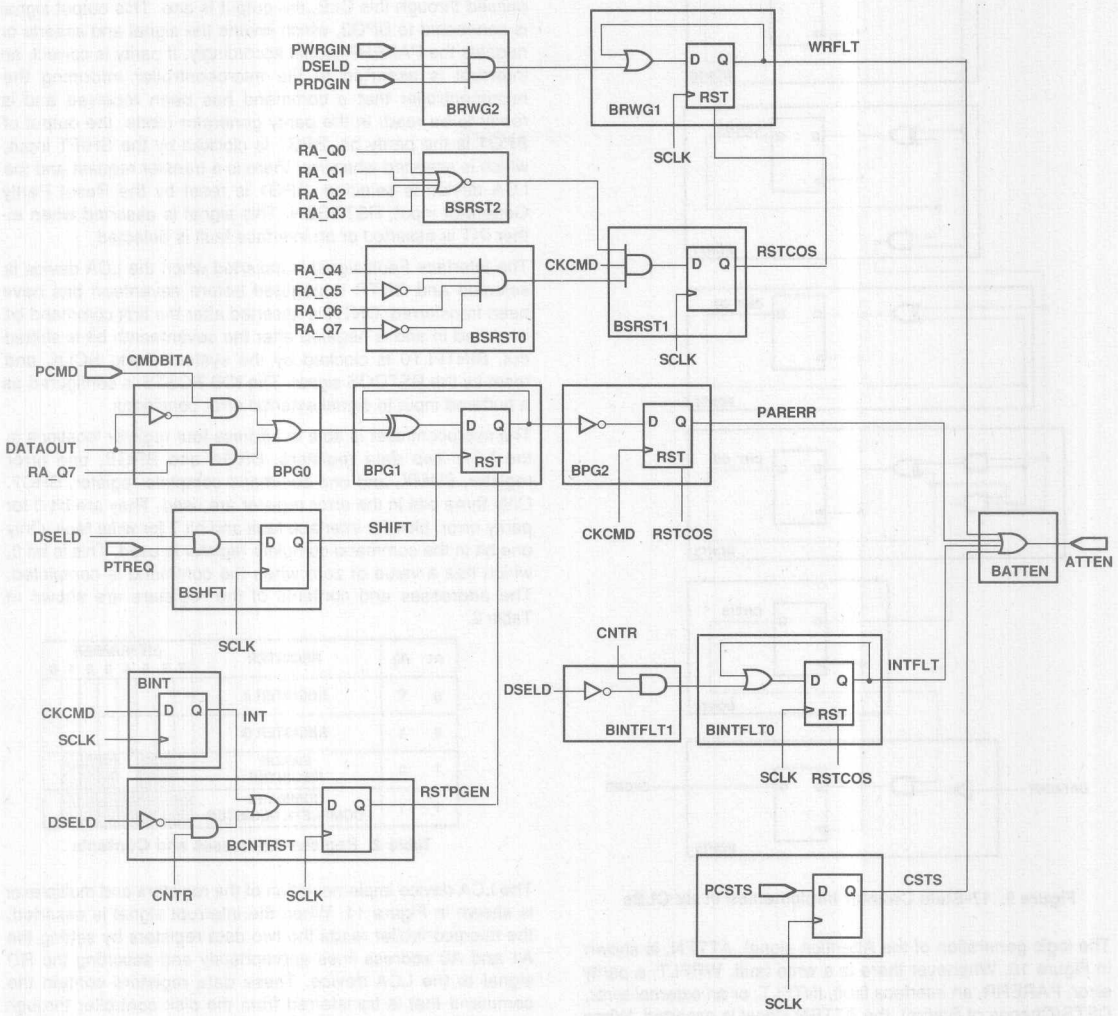


Figure 10. Generation of the Attention Signal by the Four Possible Error Conditions

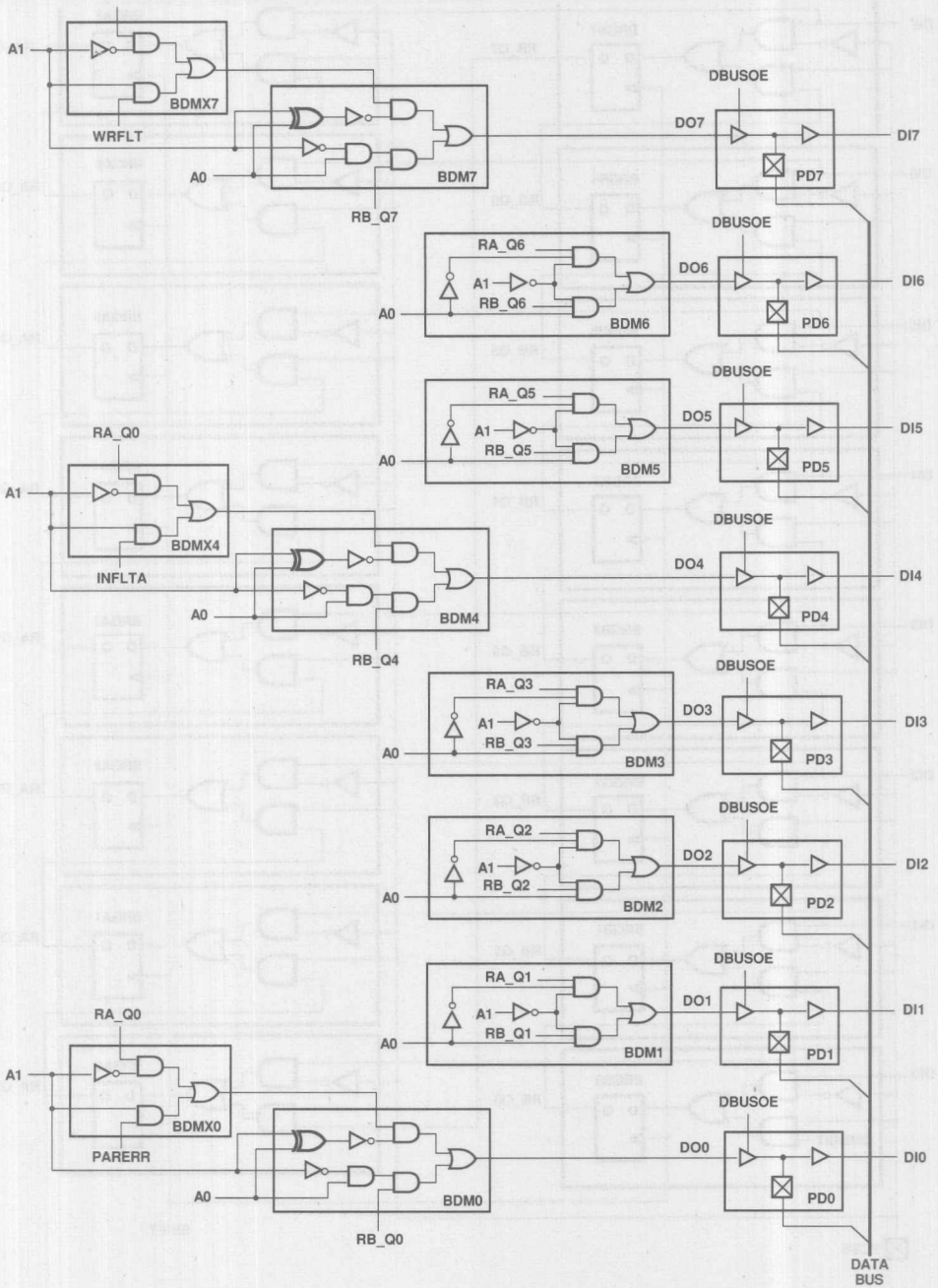


Figure 11. Registers A, B, Error Register, and Multiplexer

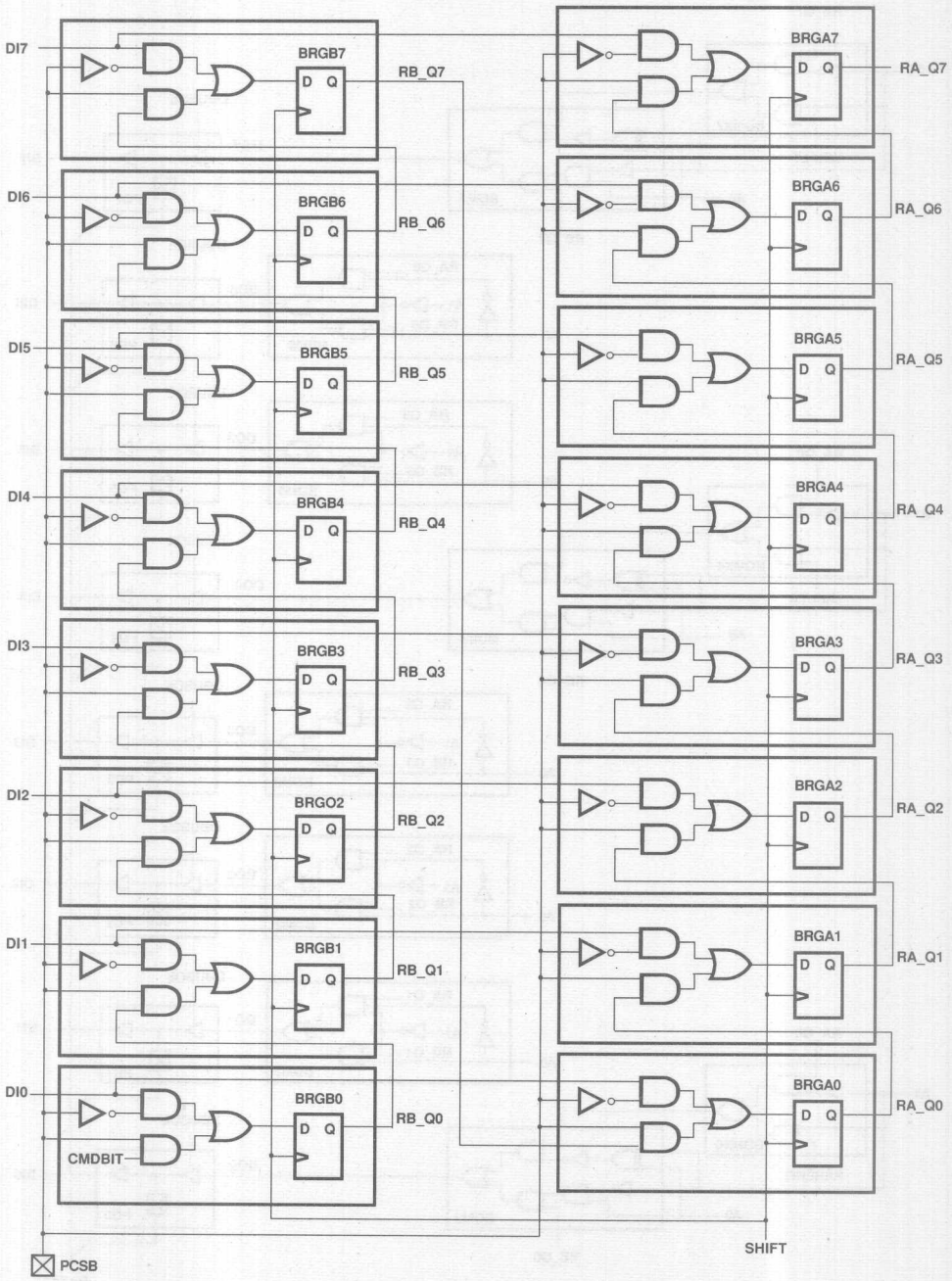


Figure 11. Registers A, B, Error Register, and Multiplexer (Continued)

Design Considerations

The circuit diagram of the programmed LCA device is shown in Figure 12. This design implementation uses sixty-three of the sixty-four CLBs within the LCA. This translates into a 98% usage. The rightmost column of CLBs contains the multiplexer, which selects between registers A, B, or the error register to be placed on the data bus. Registers A and B are placed on the third and second column from the right end, respectively. The registers and multiplexer are placed physically close to each other to simplify routing. The CLBs which execute a particular function are placed physically next to each other. These functions include the parity generator/checker, 17-state counter, read gate/write gate logic, drive selection logic and error detection logic. The read gate/write gate logic and the drive selection logic, which require much I/O activity, are placed in CLBs near the edges of the device.

Another consideration in implementing an LCA design is routing. Long lines are available for signals that have to travel a long distance within the LCA device. These lines can also be used for signals that must have minimal skew between different destinations. An automatic routing program is available in the XACT software package.

Routing requires careful consideration when a design uses a high percentage of the available CLBs (above 95%). Although automatic routing is available, designs with high CLB usage may require point-to-point routing (EDITNET command in XACT). Some of the techniques that can be used are swapping input pins, swapping CLBs, and implementing buffers in unused sections of the CLBs. When swapping input pins, the designer must be careful because certain functions, such as the clock input to the flip-flop, may only be input on certain pins. Swapping CLBs is very simple because the XACT software provides a command for swapping CLBs, but sometimes not all of the signals can be successfully rerouted. Passing a signal through a CLB presents a routing channel that is not otherwise available. However, a delay is added to the signal. Usually, a combination of these techniques can be used to successfully route a high-density design.

The high number of IOBs in the LCA device gives the designer much flexibility in designing the pinout of the device. In this design, thirty-five of the IOBs are used. The IOBs that are not being used for the actual design are not left unused, but are configured as outputs and are connected to various signals within the LCA device to provide test points for the LCA device. This greatly increases the testability of the design once it is placed on the board.

XACT is used by the designer to define the CLBs and IOBs, and to perform EDITNET. Alternately, the design may be input using the schematic capture software offered by Daisy and Futurenet. In these methods, an automatic place and route software package divides the design into blocks that may be implemented in CLBs and IOBs, and routes the CLBs and IOBs automatically. Once the design has been completed, it may be simulated using the software package P-SILOS.

Conclusion

The LCA device provides many advantages to the user. Its high gate count and I/O capability could potentially replace several PLDs in many applications, hence reducing board space. The LCA device is preferred by many customers over gate arrays because it is reprogrammable 'on-the-fly' and there are no long design cycles and initial NRE cost of a gate array. The current design file, XDES15.LCA, is available upon request. The bit pattern and the .LCA file will be provided for programming the LCA device in an EPROM.

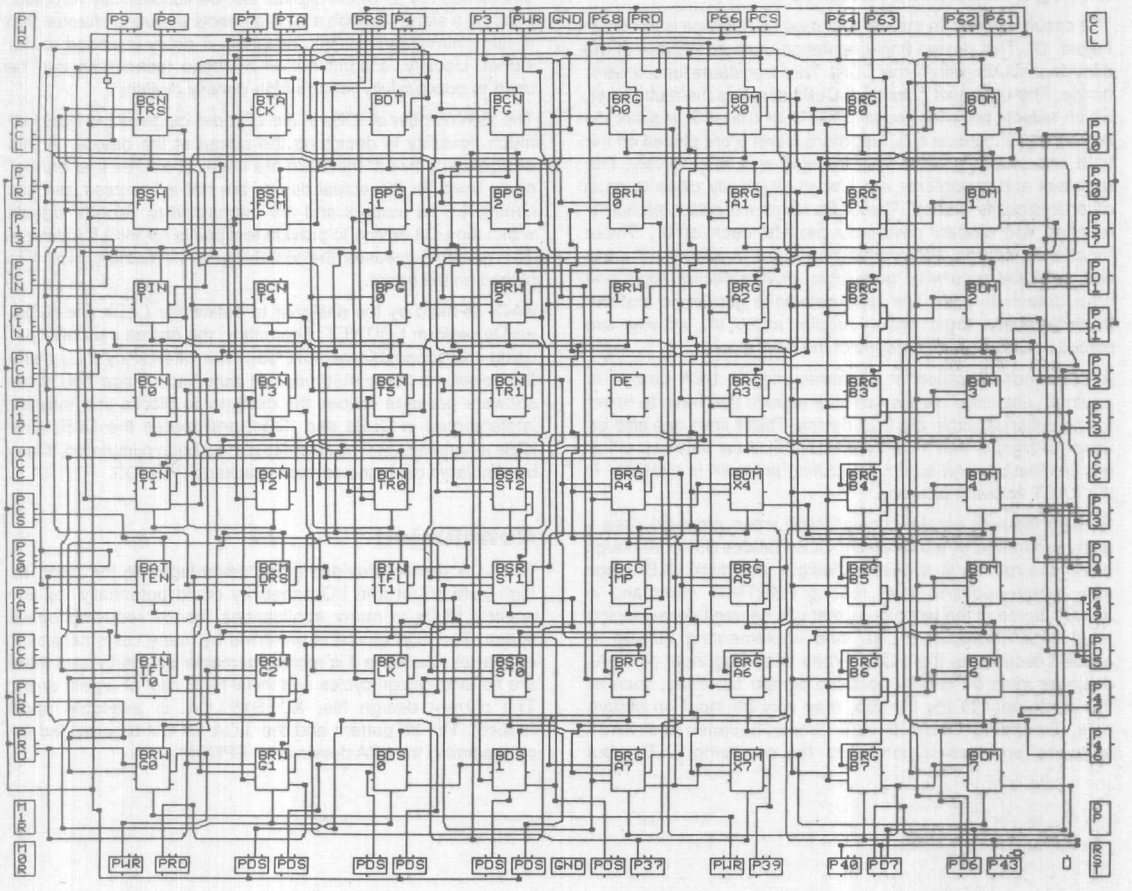


Figure 12. ESDI Translator LCA Design

Controller in PROSE

Introduction

The increasing complexity of today's computers is requiring increased use of distributed control design techniques. Such a system might include a 32-bit microprocessor as the Central Processing Unit (CPU), a 16- or 8-bit microcontroller as a peripheral controller, and a complex sequencer as a peripheral drive controller. The new PROSE (PROgrammable SEquencer) device from Monolithic Memories offers a simple, fast, and effective method for implementation of peripheral drive controllers.

This article will describe the design of a QIC-02 compatible tape drive sequencer in the PROSE device. The design requirements will be outlined, and then the design process will be described in detail for a section of the design.

QIC-02 Tape Drive

The peripheral to be controlled by our circuit is a 1/4" streaming tape drive that operates according to the QIC-02 interface standard. This standard specifies the command set and resulting tape drive control signals, as shown in Figure 1.

COMMAND	QIC-02 COMMAND*							
	D7	D6	D5	D4	D3	D2	D1	D0
Invalid Command	x	x	x	x	x	x	x	x
Read Status	0	0	1	1	1	1	1	1
Beginning of Tape	1	1	0	1	1	1	1	0
Erase Entire Tape	1	1	0	1	1	1	0	1
Initialize Tape	1	1	0	1	1	0	1	1
Write	1	0	1	1	1	1	1	1
Read	0	1	1	1	1	1	1	1
Read File Mark	0	1	0	1	1	1	1	1
Write File Mark	1	0	0	1	1	1	1	1

*The 8 bit commands shown in this table are inverted from the QIC-02 specification. This is because the PAL device has been programmed to accept active low signal inputs as defined by the QIC-02 specification.

Figure 1. QIC–02 Commands

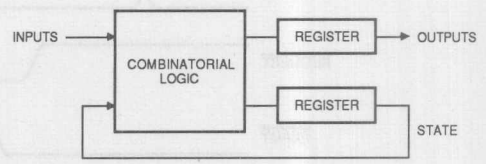
Each command requires a certain sequence of events for execution. For example, the Erase Tape command requires rewinding the tape, asserting the Erase and Initialize signals, moving the tape forward to the end, removing Erase and Initialize, and then rewinding the tape again. This type of function requires feedback of current state information, and additional inputs, for next state selection. This is the basic definition of a sequencer, or state machine (Figure 2).

Sequencers are a common design element; they include any digital device which traverses through a sequence of states in an orderly fashion. A state is a set of values (a count) measured at different parts of the circuit. State machines are often used for timing delays, control signal generation, arbitration, event monitoring, and multiple condition testing.

Our tape drive sequencer must provide the proper control signals for the tape drive and then hold them until executed, monitor the status of the tape drive, and watch for new commands from the microcontroller. The block diagram of the system is shown in Figure 3.

The peripheral microcontroller generates the 8-bit QIC-02 commands in response to the central 32-bit microprocessor. The sequencer decodes the command, determining which of the commands has been sent, and then begins execution of the command.

Handshaking is performed by a Request signal from the microcontroller, signifying that a valid command is available, and a Ready signal from the sequencer, signifying that the command has been decoded or executed and a new command can be sent (Figure 4). Upon reset, the PROSE device asserts READY, informing the peripheral controller that the tape drive is ready to accept a command. The peripheral controller asserts REQUEST after issuing a command on the eight-bit data bus. When the PROSE device detects REQUEST asserted, READY is negated and the command is decoded. After the command is decoded, READY is asserted by the PROSE device, at which time REQUEST is negated by the peripheral controller. When REQUEST is negated, READY is negated by the PROSE device.



416 02

Figure 2. General State Machine.

After the PROSE device decodes the command, the command is executed by outputting the proper drive codes and asserting the proper signals. After the command has been completed, READY is asserted by the PROSE device, informing the peripheral controller that the next command may be issued.

Command Execution

The PROSE device must be reset after power up before any command execution takes place. This is accomplished by pulsing the Preset pin low. This places the PROSE device into the Initialize state. The first command that is issued by the peripheral controller must be a Select Drive command. If the correct drive is selected, the PROSE device will enter a Drive Selected state. A Select Drive command that selects an incorrect drive or any other command will be ignored. Once in the Drive Selected state, any command issued will be executed. The drive is unselected and returned to the Initialize state by issuing a Select Drive command for a different drive.

The following text describes the execution of each of the QIC-02 commands (see QIC-02 specification for timing diagrams):

Beginning of Tape Command

The Beginning of Tape command code directs the PROSE device to output the Rewind drive code. This causes the tape drive to rewind the tape. When BOT (Beginning Of Tape) is asserted by the tape drive, the Rewind drive code is negated and READY is asserted. BOT is asserted when the Beginning of Tape hole is detected by the tape drive.

Erase Command

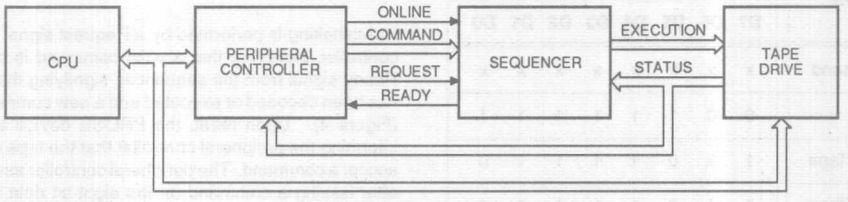
The Erase command code directs the PROSE device to output the Rewind drive code until BOT is asserted, then assert ERASE and INITIALIZE. When ERASE is asserted, the tape drive activates the erase head and moves the tape forward until EOT (End Of Tape) is asserted by the tape drive. EOT is asserted when the End of Tape hole is detected by the tape drive. When EOT is asserted, ERASE and INITIALIZE are negated and the Rewind drive code is output until BOT is asserted. When BOT is asserted, the Rewind drive code is negated and READY is asserted.

Initialize Command

The Initialize command code directs the PROSE device to assert INITIALIZE and to output the Rewind drive code. When BOT is asserted, the Rewind drive code is negated and the Fast Forward drive code is output. When EOT is asserted, the Fast Forward drive code is negated and the Rewind drive code is output. Finally, when BOT is asserted, the Rewind drive code and INITIALIZE are negated and READY is asserted.

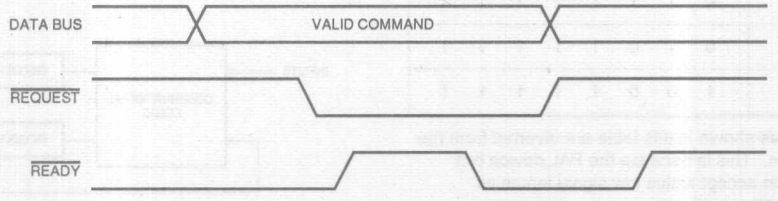
Write File Mark Command

The Write File Mark command directs the PROSE device to output the Write File Mark drive code. When FM is asserted, the Write File Mark drive code is negated and READY is asserted. FM is asserted after the file mark has been written.



416 03

Figure 3. Peripheral Controller System Including Tape Drive Sequencer.



416 04

Figure 4. Command Transfer Protocol.

Read Command

Prior to issuing the Read command, ONLINE must be asserted by the peripheral controller. If ONLINE is not asserted when the Read command code is decoded, the Invalid Command drive code will be output and the PROSE device will wait for the next command. When the Read command is issued and ONLINE is asserted, the PROSE device outputs the Read drive code. Once the Read drive code has been output, the Read command may be terminated by either the tape drive or the peripheral controller. The tape drive terminates the Read command by asserting FM. FM is asserted when a file mark is detected. When this occurs, the Read command code is negated and READY is asserted. The peripheral controller may terminate the Read command by either negating ONLINE or by issuing a Read File Mark command. When ONLINE is negated, the Read drive code is negated and the Rewind drive code is output until BOT is detected, at which time READY is asserted. When the Read File Mark command is issued, the Read drive code is negated and the Read File Mark command is executed.

Read File Mark Command

The Read File Mark command directs the PROSE device to output the Read File Mark drive code. The Read File Mark code causes the tape drive to move to the next file mark. When FM is asserted by the tape drive, the Read File Mark drive code is negated and READY is asserted. FM is asserted when a file mark is detected.

Read Status Command

The Read Status command directs the PROSE device to transfer six bytes of status data to the peripheral controller. After the command is decoded, the PROSE enables the first byte by outputting the Status Byte 1 drive code and asserts READY. The peripheral controller responds by reading the status byte and asserting REQUEST. When the PROSE device detects REQUEST asserted, READY is negated and the next status byte is enabled. READY is not asserted until after the peripheral controller negates REQUEST. The remaining five status bytes are transferred in a similar manner. After all six bytes are transferred, READY is asserted.

Write Command

We will look at the Write command in detail as an example of the design process. The Write command allows the peripheral controller to write data onto the tape. Data is transferred via the same eight-bit bus that provides the command code.

Prior to issuing the Write command, Online must be asserted by the peripheral controller. If Online is not asserted when the Write command code is decoded, the sequencer will assert the Invalid Command signal and wait for the next command.

If Online is asserted when the Write command is decoded, the sequencer will assert the Write signal for the tape drive. Data is sent from the peripheral controller to the tape drive on the eight-bit bus, and the data is written onto the tape.

The Write command may be terminated by either the tape drive or the peripheral controller. The tape drive may cause the Write command to be terminated if the end of the tape is reached. When the early warning hole is detected by the tape drive, it asserts EWH. When this happens, the sequencer must negate the Write signal and then assert the End of Media signal. It then waits to decode the next command.

The peripheral controller may terminate the Write command when all of the data has been written. This may be noted by negating the Online signal, or by asserting the Request signal. When Online is negated, the sequencer must negate the Write signal. It then outputs the Write File Mark signal until the file mark is activated by the tape drive. Once the file mark is activated, the sequencer must rewind the tape.

When the Request signal is asserted by the peripheral controller, the sequencer negates the Write signal. It then decodes the new command, which usually would be a Write File Mark command.

State Diagram Representation

All of this complex interaction can be described in a simple state diagram. State diagrams are the universal language for describing state machines. The diagrams depict the two major operations in a state machine: control sequencing and output generation. Control sequencing is changing from present state to next state. It can be direct, based only upon present state, or conditional, based upon present state and input conditions. Similarly, output generation can be direct, based upon present state only, or conditional, based upon present state and input conditions. The former is called a Moore state machine, while the latter is a Mealy state machine.

Transitions for our tape drive sequencer depend on the state and conditional inputs, as with the Request and Online inputs during execution of the Write command. Outputs, however, will depend only on the present state; thus, our sequencer will be a Moore state machine.

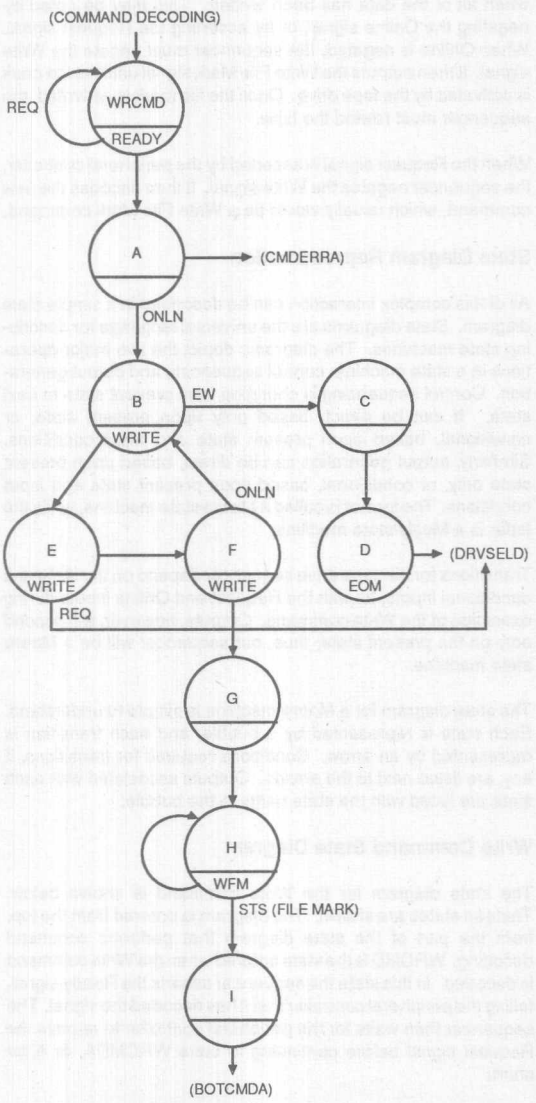
The state diagram for a Moore machine is simple to understand. Each state is represented by a bubble, and each transition is represented by an arrow. Conditions required for transitions, if any, are listed next to the arrows. Outputs associated with each state are listed with the state name in the bubble.

Write Command State Diagram

The state diagram for the Write command is shown below. Thirteen states are shown. The diagram is entered from the top, from the part of the state diagram that performs command decoding. WRCMD is the state entered when the Write command is decoded. In this state the sequencer asserts the Ready signal, telling the peripheral controller that it has decoded the signal. The sequencer then waits for the peripheral controller to remove the Request signal before continuing to state WRCMDA, or A for short.

In state A the sequencer immediately checks to make sure that Online is asserted before beginning the Write sequence. If it is not, the Invalid Command signal is output and the sequencer enters another part of the state diagram (CMDERRA).

If Online is asserted, the sequencer moves to state B, asserting the Write signal to the tape drive to allow writing to begin. The sequencer moves from states B, E, and F and back to B, until the Write process is interrupted. The tape drive responds by moving forward and writing the information provided on the eight-bit data bus.



416 05

Figure 5. Write Command State Diagram.

As described, the Write command can be terminated by either the tape drive or the peripheral controller. State B checks for assertion of the EWH (Early Warning Hole) signal from the tape drive. If asserted, the sequencer immediately negates the Write signal in state C, and then asserts the EOM (End of Media) signal to the peripheral drive controller in state D. The sequencer then goes to the Drive Selected state (DRVSELD) and begins the command decoding procedure in another part of the state diagram.

State E checks for assertion of the Request signal, signalling the end of the data to be written. In this case, the sequencer would immediately negate the Write signal and go to the Drive Selected state.

State F checks for removal of the Online signal. If Online is true, the sequencer moves back to state B and continues writing. If not, it will move to state G, removing the Write signal, and then state H, outputting the Write File Mark signal. It then waits in state H until the File Mark signal is activated by the tape drive, indicating that the file mark has been written. When File Mark is activated, the Write File Mark signal is removed in state I, and the sequencer moves to the Beginning of Tape command sequence to rewind the tape.

Device Selection

So far, the design has been described and defined without regard to the implementation methodology. The complete sequencer can be implemented in several medium-size discrete logic devices, or one large custom circuit.

The best choice is often a programmable logic device. Programmable logic provides the integration of custom logic with the quick time-to-market of standard products. Especially important in this application is the ability to adapt to design changes. The QIC-02 specification can be modified to enhance performance or add features. Of course, any design errors can also be quickly fixed, even after layout of the PC board.

Programmable logic devices offer the complexity to implement large designs in few devices. At the same time, software support tools allow very simple design entry, even of complex designs. This design will require almost no additional work to describe for PLD software. Device programmers provide an instant custom device, and can even test the device immediately after programming.

Selection of PROSE Device

The PROSE device is selected as the heart of this application. The primary reason is the number of states required for implementation of the sequencer. We saw that the Write command alone had thirteen states, although some were transitions to other parts of the state diagram. There are eight other commands required, totaling over 100 states. Only the PROSE device provides enough states in a single programmable chip.

Another requirement in many applications is high speed. The PROSE device offers a maximum frequency of 30 MHz internally. At the same time, the PROSE device has enough logic on board to require very few surrounding logic chips.

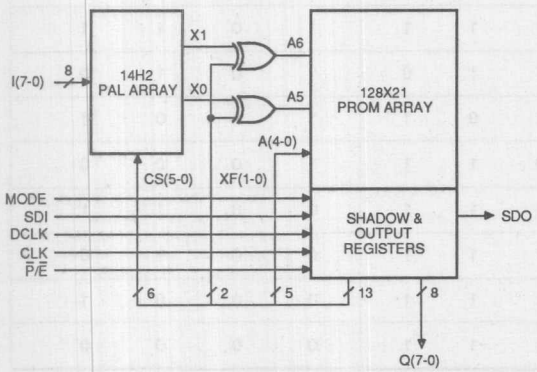
The PROSE device has the architecture shown in Figure 6. It is based on a 128x21 PROM array, providing 128 states of 21 bits each. The 128 states will include the states WRCMD and its following states from A through I. Eight of the twenty-one bits in each state are the outputs of the device, providing signals such as the Write and Ready signals.

The other thirteen bits in each state feed back internally for next state selection. Six of the feedback signals feed back into a PAL array, which is also fed by eight external inputs. These external inputs are the condition inputs, such as the Request and Online signals, which determine the next state. The feedback signals determine which of these eight inputs will determine the next branch.

The outputs of the PAL array are two address lines to the PROM array. Before reaching the PROM array, however, these signals go through two Exclusive-OR (XOR) gates controlled by feedback signals. These gates are used by the software to optimize the fit of the design within the device, and will be described later. The other five address lines to the PROM array are fed back directly from the previous state. Thus, two of the lines can be changed by input conditions, providing four-way branching capability from any state.

Expanding Inputs for the PROSE Device

The PROSE device has a balanced architecture of eight inputs and eight outputs. However, we already know that we need more than eight inputs in the design. The eight bits of the command code from the peripheral controller need to be decoded, along



416 06

Figure 6. PMS14R21 Architecture

and File Mark signals already mentioned. In addition to these signals, the sequencer must monitor the End of Tape and Beginning of Tape signals from the tape drive, making a total of fourteen inputs.

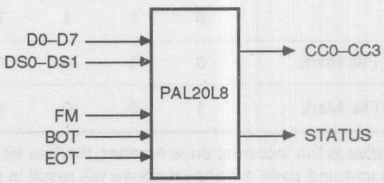
A simple solution is encoding the eight different eight-bit commands into a three-bit command code before reaching the PROSE device. This can be easily done in almost any 20-pin PAL device. With a PAL device as the solution, we can easily add even more logic surrounding the sequencer, including drive selection. Drive selection logic requires that more commands be added to the original eight, forcing the use of four bits for the encoded commands. Three of the independent status signals from the tape drive can also be combined in the PAL device (Beginning of Tape, End of Tape, and File Mark), providing one Status signal. (Early Warning Hole remains a separate input.)

The resulting logic block fits easily into a simple PAL20L8 combinatorial 24-pin PAL device (Figure 7). Several versions of this device are available to choose from, including a CMOS device offering zero standby power, and high-speed devices at 15 nanoseconds or less propagation delay.

The encoding of the command codes, including drive selection, is shown in Figure 8. Now we must write the equations for PALASM 2 software for implementation in a PAL20L8. Writing the equations for each output is a simple matter; active-low equations are written where each product term is one of the commands in which that particular command code bit is zero. For example, the equation for CC3 is simply the sum of the products defining the Write, Read, Read File Mark, and Write File Mark commands. The equation would be:

```
/CC3   =  D7*/D6* D5* D4* D3* D2* D1* D0 (Write)
       + /D7* D6* D5* D4* D3* D2* D1* D0 (Read)
       + /D7* D6*/D5* D4* D3* D2* D1* D0 (Read
       File Mark)
       +  D7*/D6*/D5* D4* D3* D2* D1* D0 (Write
       File Mark)
```

where / means inversion, * means AND, and + means OR. The equations can be minimized either by observation or by running them through the program MINIMIZE in the PALASM 2 software suite. The full PALASM 2 file can be simulated for design verification, and then processed to create a JEDEC file for programming.



416 07

Figure 7. PAL Encoding Logic Block.

Expanding Outputs for the PROSE Device

The section of the design that we looked at in detail required five output signals: Ready, Invalid Command, Write, End of Media, and Write File Mark. Other required outputs include Read, Read File Mark, Fast Forward, Rewind, Erase, Initialize, and Drive Selected. These twelve outputs already extend us beyond the eight outputs available in the PROSE device. In addition, six Enable Status bits are required, making a total of eighteen outputs required.

A simple solution can again be provided by a programmable logic device, to decode outputs from the PROSE device into the required output signals. The requirement for many outputs is met by two PAL devices, the PAL6L16 and PAL8L14. However, even faster solutions are provided by two PLE (PROM as Logic Element) devices, the PLE5P16 and PLE6P16. These devices are specialized PROMs, both with sixteen outputs. The PLE5P16 offers the highest speed at 18 ns propagation delay, and provides an output enable, which can be useful when testing the tape drive independent of the sequencer logic.

In the PLE5P16 we want to decode four outputs from the PROSE device into fourteen signals. The four signals to be left as direct outputs from the PROSE device can be the signals used more often, or those that require higher speed. In this case, Ready, Drive Selected, Erase, and Initialize are left direct, while the other signals are decoded through the PLE device. The equations for the decoding are arbitrary, as long as every signal in asserted by its own code. (None of these signals are to be asserted at the same time.) The decoding is shown below.

The generation of the equations is similar to that shown previously. For example,

```
STS0 = /CD3*/CD2*/CD1* CD0
```

is the decoding for asserting the Enable Status byte 0 signal.

However, this time PLEASM software is used for design entry. The design method is equivalent to that for PAL devices in PALASM software, but the software creates a programming file in the PROM format.

COMMAND	QIC-02 COMMAND								PAL COMMAND CODE			
	D7	D6	D5	D4	D3	D2	D1	D0	CC3	CC2	CC1	CC0
Invalid Command**	x	x	x	x	x	x	x	x	1	1	1	1
Select Drive 1*	1	1	1	1	1	1	1	0	1	1	1	0
Select Drive 2*	1	1	1	1	1	1	0	1	1	1	1	0
Select Drive 3*	1	1	1	1	1	0	1	1	1	1	1	0
Select Drive 4*	1	1	1	1	0	1	1	1	1	1	1	0
Read Status	0	0	1	1	1	1	1	1	1	0	1	1
Beginning of Tape	1	1	0	1	1	1	1	0	1	0	1	0
Erase Entire Tape	1	1	0	1	1	1	0	1	1	0	0	1
Initialize Tape	1	1	0	1	1	0	1	1	1	0	0	0
Write	1	0	1	1	1	1	1	1	0	0	1	1
Read	0	1	1	1	1	1	1	1	0	0	1	0
Read File Mark	0	1	0	1	1	1	1	1	0	0	0	1
Write File Mark	1	0	0	1	1	1	1	1	0	0	0	0

* If the drive is the incorrect drive number, the four bit command code is 1101
** Any command code not shown above will result in an illegal command code.

Figure 8. PAL Command Codes Table.

CD3	CD2	CD1	CD0	SIGNAL ASSERTED
0	0	0	0	No Command
0	0	0	1	STS0 (Enable Status byte 0)
0	0	1	0	STS1 (Enable Status byte 1)
0	0	1	1	STS2 (Enable Status byte 2)
0	1	0	0	STS3 (Enable Status byte 3)
0	1	0	1	STS4 (Enable Status byte 4)
0	1	1	0	STS5 (Enable Status byte 5)
0	1	1	1	EOM (End of Media)
1	0	0	0	RFM (Read File Mark)
1	0	0	1	WFM (Write File Mark)
1	0	1	0	READ (Read Enable)
1	0	1	1	WRITE (Write Enable)
1	1	0	0	FFWD (Fast Forward)
1	1	0	1	RWD (Rewind)
1	1	1	0	ICMD (Invalid Command)
1	1	1	1	No Command

Figure 9. PLE Device Decoding

PROSE Design Specification

We are now ready to discuss the details of the design specification for the PROSE device. The software used is a part of the PALASM 2 software suite called PROASM for PROSE Assembler. It requires almost no additional work on the part of the designer; it simply requires a description of the state diagram. PROASM has the following file format:

- Declaration Section
- State Section
- Conditions Section

Declaration Section

The Declaration Section is the location of the general information about the design, including the design engineer's name and company. It provides for device selection; in this case, we are using the PROSE device, which has the part number PMS14R21. It also allows the definition of all of the inputs and output signal names. Remember, four of the inputs are PAL device outputs (CC0-CC3), and four of the outputs feed inputs to the PLE device (CD0-CD3).

```
TITLE       QIC-02 COMMAND DECODER
PATTERN     1
REVISION    B
AUTHOR KEN  WON
COMPANY     MMI
DATE        JANUARY 19, 1987

CHIP        QIC_02_CMD_DEC PMS14R21

;INPUT PIN  1   2     3       4
            CLK DCLK /ONLINE /REQUEST

;INPUT PIN  5    6    7    8    9      10
            /CC0 /CC1 /CC2 /CC3 STATUS EWH
;INPUT PIN  11  12
            SDI GND

;OUTPUT PIN 13     14  15  16  17  18
            PRESET SDO CD3 CD2 CD1 CD0

;OUTPUT PIN 19         20    21     22
            INITIALIZE ERASE DRSELD READY
;OUTPUT PIN 23   24
            MODE VCC
```

State Section

The next part of the design file is the State section. This section begins with the keyword STATE. The first item required is selection of either a Moore or a Mealy type state machine. As discussed previously, a Moore type state machine is required.

```
STATE
MOORE_MACHINE
```

Next, several general or default values are specified. First is selection of the programmable Preset/Enable function pin. Either can be used: in this case, Preset is desired, so the keyword MASTER_RESET is entered.

```
MASTER_RESET
```

Next are global default output values. Global default output values are the values to be assigned to the outputs when the design specification does not specify a particular value. This is not used, so no default output function is entered. However, the design file can be simplified by specifying a default value for the outputs (either High or Low), and then only specifying the output value when it differs from the default value.

Next are global default transitions or branches. Global default branches are the branches to be taken when no branch is specified in the design file. This is a method of not only simplifying the design file, but also guaranteeing that there is a transition from every state for every input condition. In this design, the default is chosen to be HOLD_STATE; this means that the sequencer will remain in the same state if no other transition is enabled.

```
DEFAULT_BRANCH HOLD_STATE
```

Next is the specification of the state to enter after power-up. The PROSE device has power-up preset, whereby all twenty-one flip-flops power up into the logic One state. This state is called

POWER_UP. The transition from this state (on the first clock edge) shows how transition equations are written. The format is

```
STATE_X := CONDITION_A -> STATE_Y
```

meaning that when in State X, if condition A is true the machine will transition to state Y. In this design, we want to transition to state INIT on any condition, so the equation is written

```
POWER_UP := VCC -> INIT
```

where VCC means unconditionally.

The first transition in the Write command state diagram requires two branches. The local default branch is indicated by the + sign, which is the "otherwise" operator. Thus, from state WRCMD, we have

```
WRCMD := REQ -> WRCMD
         +-> WRCMDA
```

which means "if REQ (Request) is true, go to state WRCMD (stay in the same state), otherwise (if REQ is not true) go to state WRCMDA."

The remaining transition equations are written similarly. In effect, the arrows in the state diagram are directly transcribed into arrows in the equations.

```
WRCMDA := ONLN -> WRCMDB
          +-> CMDERRA

WRCMDB := EW   -> WRCMDC
          +-> WRCMDE

WRCMDC := VCC  -> WRCMDD

WRCMDD := VCC  -> DRVSELD

WRCMDE := REQ  -> DRVSELD
          +-> WRCMDF

WRCMDF := ONLN -> WRCMDB
          +-> WRCMDG

WRCMDG := VCC  -> WRCMDH

WRCMDH := STS  -> WRCMDI
          +-> WRCMDH

WRCMDI := VCC  -> BOTCMDA
```

Note that the equation for WRCMDH does not need the second line; the global default transition is to stay in the same state, so it does not need to be specified here.

Output Definition

The next task is to define the eight outputs for each of these states. Outputs are defined in the form of an equation. For example,

```
STATE_X.OUTF := OUT_1 * /OUT_2
```

means that in state X, output 1 will be High and output 2 will be Low. If other outputs exist, they will be assigned their default values.

Output equations for the Write command section are written according to the state diagram. In all of the Write states, Drive Selected (DRVSELD) remains active. Thus, in state WRCMD only Drive Selected and the Ready signal are active, telling the peripheral controller that the Write command has been decoded and is ready to be executed. The resulting output equation for state WRCMD is

```
WRCMD.OUTF :=   READY*/CD3*/CD2*/CD1*/CD0
              */ERASE*/INITIALIZE* DRSELD
```

The other states have similar output equations, translating the required output signal to the appropriate encoding for the PLE device to decode.

```
WRCMDA.OUTF := /READY*/CD3*/CD2*/CD1*/CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDB.OUTF := /READY* CD3*/CD2* CD1* CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDC.OUTF := /READY*/CD3*/CD2*/CD1*/CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDD.OUTF := /READY*/CD3* CD2* CD1* CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDE.OUTF := /READY* CD3*/CD2* CD1* CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDF.OUTF := /READY* CD3*/CD2* CD1* CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDG.OUTF := /READY*/CD3*/CD2*/CD1*/CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDH.OUTF := /READY* CD3*/CD2*/CD1* CD0
               */ERASE*/INITIALIZE* DRSELD

WRCMDI.OUTF := /READY*/CD3*/CD2*/CD1*/CD0
               */ERASE*/INITIALIZE* DRSELD
```

Again, if default output values were used, these equations could be extremely simplified.

Conditions Section

The final section is the Conditions section, designated by the keyword Conditions. This section defines the conditions used in the state equations. In this design, the conditions are simply the inputs to the PROSE device, and are straightforward:

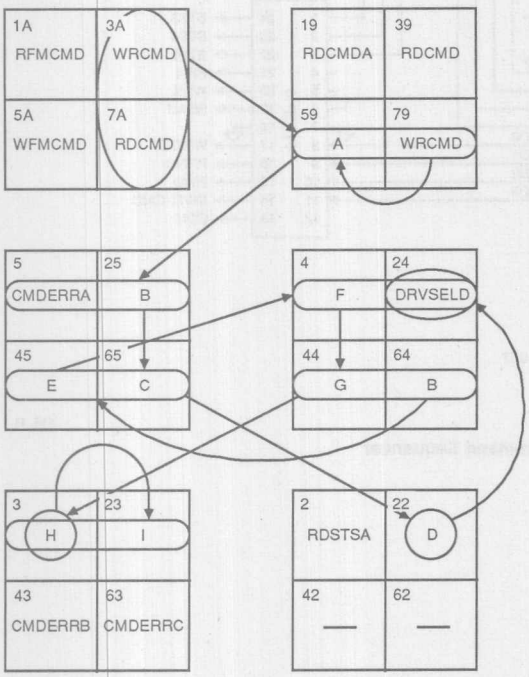
```
CONDITIONS

ONLN = ONLINE
STS = STATUS
EW = EWH
REQ = REQUEST
```

Other conditions used in the design file are also placed in this Conditions section.

The other sections of the design file are completed in a similar manner, and the whole design is assembled by PROASM software. A key feature of PROASM software, in addition to design verification, is automatic state assignment. State locations in the PROM and conditional product terms in the PAL array are used to the greatest possible efficiency, to provide the best fit of the logic within the device.

The automatic fitting is easily seen in Figure 10. Each individual state can branch to one of four locations: these four locations form a "primary location." For example, the hex addresses 1A, 3A, 5A, and 7A form a primary location, which can be designated by the smallest address, 1A.

The software optimizes use of these primary locations to eliminate excessive duplication of states in different primary locations. For example, the state WRCMD is duplicated in primary location



416 10

Figure 10. State Assignments for the Six Primary PROM Locations Used for the Write Command Sequence. Arrow Indicate Potential State Transitions, to a Set of Circled Next States. All Next States Must be in the Same Primary Location.

WRCMD itself or WRCMDA (or A in the shortened form). The branch cannot be made to the primary location where WRCMD already exists, because the other three locations have already been used. Thus, WRCMD must be duplicated in another primary location which has an open location for state A.

The software then puts the two-way branch from state A in primary location 5. It then makes use of the other half of the primary location for the two-way branch from state B. However, when it then assigns the two-way branch from state F to states B or G, it must duplicate B in primary location 4 since there is no room left in its primary location (5) for G.

Note that the software avoids duplication of state DRVSELD, however. The two-way branch from state E to states DRVSELD or F is placed in the other half of primary location 4. When DRVSELD is branched to again from state D, the same location can be used.

This duplication avoidance is where the XOR gates are used. To place a branch in the lower order half of a primary location, the branch addresses must be 00 or 01. This is accomplished by making the XOR feedback 00 and allowing a PAL product term to determine whether the second address bit should be a 1. To use the higher order half of a primary location (branch addresses 10 or 11), the XOR feedback would be made 10 and again a PAL product term would be used to determine the other bit. XOR feedback is also used to determine "right or left side" of a primary location, or to use a product term that already exists on one side of the PAL array.

The PROASM software provides detailed implementation results, showing how the states and product terms were assigned for every state and transition.

PROSIM Simulation

The entire sequencer can be simulated by PROSIM software, to verify the design. This can be done before any parts are programmed. Simulation results can be added to the programming file to provide automatic testing after programming.

Completed Design

The complete design required less than a week to implement from scratch. Two days were required to define the application. Only one day was required to write the PROSE design specification. Simulation and design checking took an additional day. Only half a day each were required for the PAL and PLE device designs. This included revisions to the PAL and PROSE devices to fix minor design errors. The complete design is shown in Figure 11, and the design files are shown on the following pages.

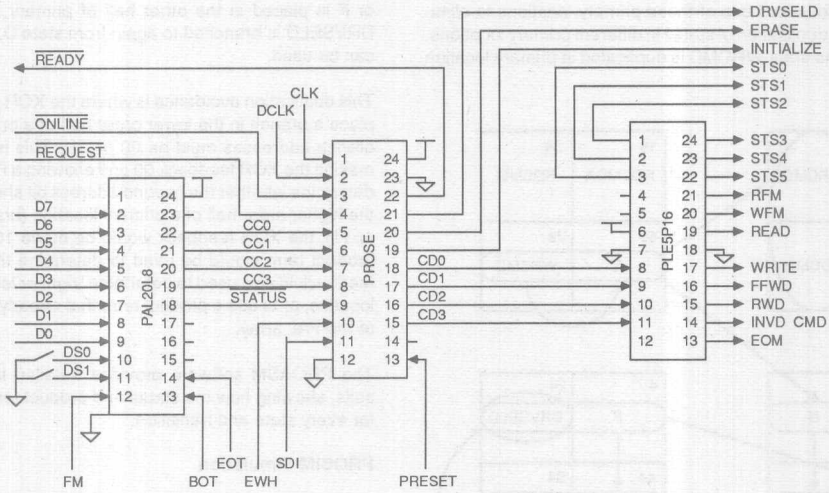
Conclusion

Programmable logic devices were once the glue logic that tied together the intelligence of a digital system. Today, advanced chips such as the PROSE device are capable of implementing distributed control functions and make up the heart of the control of the system. With complementary advanced design tools available with PALASM 2 software, PLDs are an effective approach to even the most complex tasks.

The PROSE device has been designed to implement a complex tape drive sequencer. Design implementation in the PROSE device is made easy for several reasons, including:

1. Design file requires merely transcription of the state diagram
2. Software automatically optimizes fit
3. Programmability of the device allows design errors to be fixed quickly

As a result, the complete tape drive sequencer required only four days to design and implement.



416 11

Figure 11. QIC-02 Command Sequencer

```
; THIS PAL DEVICE IS PART OF THE QIC-02 COMMAND SEQUENCER
; DESIGN. THE PRIMARY PURPOSE OF THIS PAL DEVICE IS TO ENCODE
; 8 BIT COMMANDS INTO 4 BIT COMMAND CODES.  IT IS ALSO USED TO
; ENCODE TAPE DRIVE STATUS SIGNALS AND SELECT THE DRIVE NUMBER.

TITLE           QIC-02 COMMAND DECODER PAL
PATTERN
REVISION        B
AUTHOR          KEN WON
COMPANY         MONOLITHIC MEMORIES
DATE            JANUARY 26, 1987

CHIP  QIC_02_CMD_PAL  PAL20L8

D7 D6 D5 D4 D3 D2 D1 D0 DS0 DS1 FM GND
BOT EOT NC /DSELD /CC2A /STATUS CC3 CC2 CC1 CC0 NC VCC

EQUATIONS

STATUS  = /EOT*/BOT*/FM                    ;ASSERT STATUS TO PMS14R21

/CC0    = D7*D6*/D5*D4*D3*D2*D1*/D0        ; BOT COMMAND
        + D7*D6*/D5*D4*D3*/D2*D1*D0        ; INITIALIZE
        + D7*/D6*/D5*D4*D3*D2*D1*D0        ; WRITE FILE MARK
        + /D7*D6*D5*D4*D3*D2*D1*D0         ; READ
        + D7*D6*D5*D4*DSELD                ; SELECT DRIVE

/CC1    = D7*D6*/D5*D4*D3*D2*/D1*D0        ; ERASE
        + D7*D6*/D5*D4*D3*/D2*D1*D0        ; INITIALIZE
        + D7*/D6*/D5*D4*D3*D2*D1*D0        ; WRITE FILE MARK
        + /D7*D6*/D5*D4*D3*D2*D1*D0        ; READ FILE MARK
        + D7*D6*D5*D4*/DSELD               ; UNSELECT DRIVE

/CC2    = D7*D6*/D5*D4*D3*D2*D1*/D0        ; BOT COMMAND
        + D7*D6*/D5*D4*D3*D2*/D1*D0        ; ERASE
        + D7*D6*/D5*D4*D3*/D2*D1*D0        ; INITIALIZE
        + D7*/D6*D5*D4*D3*D2*D1*D0         ; WRITE
        + D7*/D6*/D5*D4*D3*D2*D1*D0        ; WRITE FILE MARK
        + /D7*D6*/D5*D4*D3*D2*D1*D0        ; READ FILE MARK
        + /D7*D6*D5*D4*D3*D2*D1*D0         ; READ
        + /D7*/D6*D5*D4*D3*D2*D1*D0        ; READ STATUS
                                           ; (MINIMIZATION REDUCES
                                           ; PRODUCT TERMS)
/CC3    = D7*/D6*D5*D4*D3*D2*D1*D0         ; WRITE
        + D7*/D6*/D5*D4*D3*D2*D1*D0        ; WRITE FILE MARK
        + /D7*D6*D5*D4*D3*D2*D1*D0         ; READ
        + /D7*D6*/D5*D4*D3*D2*D1*D0        ; READ FILE MARK

DSELD   = /D0*/DS1*/DS0*D1*D2*D3           ; DRIVE 1 SELECT
        + /D1*/DS1*DS0*D0*D2*D3            ; DRIVE 2 SELECT
        + /D2*DS1*/DS0*D0*D1*D3            ; DRIVE 3 SELECT
        + /D3*DS1*DS0*D0*D1*D2             ; DRIVE 4 SELECT

SIMULATION

; THIS SIMULATION FILE TESTS THE OUTPUT FOR ALL VALID AND ONE INVALID
; COMMAND INPUT

TRACE_ON D7 D6 D5 D4 D3 D2 D1 D0 FM EOT BOT CC0 CC1 CC2 CC3 /STATUS /DSELD
SETF D7 D6 /D5 D4 D3 D2 D1 /D0             ;BEGINNING OF TAPE
SETF D7 D6 /D5 D4 D3 D2 /D1 D0             ;ERASE
SETF D7 D6 /D5 D4 D3 /D2 D1 D0             ;INITIALIZE
SETF D7 /D6 D5 D4 D3 D2 D1 D0              ;WRITE
SETF D7 /D6 /D5 D4 D3 D2 D1 D0             ;WRITE FILE MARK
SETF /D7 D6 D5 D4 D3 D2 D1 D0              ;READ
SETF /D7 D6 /D5 D4 D3 D2 D1 D0             ;READ FILE MARK
SETF /D7 /D6 D5 D4 D3 D2 D1 D0             ;READ STATUS
SETF /D7 D6 D5 D4 /D3 /D2 D1 D0            ;INVALID COMMAND
SETF EOT                                   ;ASSERT EOT
SETF /EOT                                  ;NEGATE EOT
SETF BOT                                   ;ASSERT BOT
SETF /BOT                                  ;NEGATE BOT
SETF FM                                    ;ASSERT FM
SETF /FM                                   ;NEGATE FM
SETF DS1 /DS0                              ;SET DRIVE NUMBER TO 3
SETF D7 D6 D5 D4 D3 /D2 D1 D0              ;SELECT DRIVE 3
SETF D7 D6 D5 D4 D3 D2 /D1 D0              ;UNSELECT DRIVE 3
TRACE_OFF
```

```
;THIS IS A SECOND REVISION OF AN APPLICATION FOR THE PROSE DEVICE.
;THE ORIGINAL APPLICATION IS CALLED 14R21.APP.  THIS APPLICATION
;IMPLEMENTS A QIC-02 COMMAND DECODER USING A PROSE DEVICE, A PAL
;DEVICE, AND A PLE DEVICE.  THE PROSE DEVICE FILE IS CALLED
;CMD_PAL.APP AND THE PLE DEVICE FILE IS CALLED CMD PLE.APP.

TITLE QIC-02 COMMAND DECODER
PATTERN
REVISION        B
AUTHOR          KEN WON
COMPANY         MMI
DATE            JANUARY 19, 1987

CHIP    QIC02_CMD_DEC   PMS14R21

CLK DCLK /ONLINE  /REQUEST /CC0 /CC1 /CC2 /CC3 STATUS EWH SDI GND
PRESET SDO CD3 CD2 CD1 CD0 INITIALIZE ERASE DRSELD READY MODE VCC
STATE
MOORE_MACHINE
MASTER_RESET
DEFAULT_BRANCH HOLD_STATE
POWER_UP  :=VCC->INIT

;@@@@@@ EQUATIONS @@@@@@

;****** DRIVE SELECTION ******
INIT            := REQ          -> DRVSELDA     ;IF /REQUEST IS TRUE, CHECK
                                                ; FOR DRIVE SELECTION
                                +-> INIT        ;OTHERWISE GOTO INIT
DRVSELDA        := COND0         -> DRVSELDB
                                +-> INIT
DRVSELDB        := COND1         -> INIT
                                +-> DRVSELDC
DRVSELDC        := COND2         -> INIT
                                +-> DRVSELDD
DRVSELDD        := COND3         -> INIT
                                +-> DRVSELDE    ;DRIVE IS SELECTED
DRVSELDE        := REQ           -> DRVSELDE    ;WAIT FOR REQ DEACTIVATED
                                +-> DRVSELDF    ;THEN GOTO DRVSELDF
DRVSELDF        := VCC           -> DRVSELD     ;DRIVE SELECTION COMPLETE

INIT.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*/DRSELD
DRVSELDA.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*/DRSELD
DRVSELDB.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*/DRSELD
DRVSELDC.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*/DRSELD
DRVSELDD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*/DRSELD
DRVSELDE.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*/DRSELD
DRVSELDF.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*/DRSELD
DRVSELD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
;****** COMMAND DECODING ******
DRVSELD         := REQ          -> CMDDECA      ;IF /REQUEST IS TRUE,
                                                ; DECODE COMMAND
                                +-> DRVSELD     ;OTHERWISE GOTO DRVSELD
CMDDECA         := COND3         -> RDWRCMD     ;IF CC3 IS TRUE, GOTO
                                                ; READ/WRITE DECODING
                                +-> CMDDECB     ;OTHERWISE CONTINUE
                                                ; COMMAND DECODING
RDWRCMD         := COND2         -> RDWRCMDA    ;IF CC2 IS TRUE, GOTO
                                                ; READ/WRITE DECODING
                                +-> CMDERR      ;OTHERWISE GOTO CMDERR
RDWRCMDA        := COND1         -> FMCMD       ;IF CC1 IS TRUE, COMMAND
                                                ; IS A FILE MARK COMMAND
                                +-> RDWRCMDB    ;OTHERWISE READ OR WRITE
RDWRCMDB        := COND0         -> RDCMD       ;IF CC0 TRUE, COMMAND READ
                                +-> WRCMD       ;OTHERWISE COMMAND IS WRITE
FMCMD           := COND0         -> WFMCMD      ;IF CC0 IS TRUE, COMMAND IS
                                                ; WRITE FILE MARK
                                +-> RFMCMD      ;OTHERWISE COMMAND IS READ
                                                ; FILE MARK
CMDDECB         := COND2         -> CMDDECC     ;IF CC2 IS TRUE, CONTINUE
                                                ; COMMAND DECODING
                                +-> DESELDA     ;OTHERWISE CHECK FOR
                                                ; DESELECTING DRIVE
CMDDECC         := COND1         -> CMDDECD     ;CONTINUE DECODING COMMAND
                                +-> CMDDECE     ;CONTINUE DECODING COMMAND
CMDDECD         := COND0         -> INITCMD     ;IF CC0 TRUE, COMMAND INIT
                                +-> ERASECMD    ;OTHERWISE COMMAND IS ERASE
CMDDECE         := COND0         -> BOTCMD      ;IF CC0 TRUE, COMMAND BOT
                                +-> RDSTSCMD    ;OTHERWISE COMMAND IS READ
                                                ; STATUS
```

```
CMDDECA.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
CMDDECB.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
CMDDECC.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
CMDDECD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
CMDDECE.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDWRCMD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDWRCMDA.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDWRCMDB.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
FMCMD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD

;****** DESELECT DRIVE ******
DESELDA          := COND1         -> DESELDB       ;CONTINUE CHECKING FOR
                                                   ; DESELECT COMMAND
                                 +-> DESELDC       ;OTHERWISE GOTO DESELDA
DESELDB          := COND0         -> CMDERR        ;IF CC0 IS TRUE, COMMAND
                                                   ; IS INVALID
                                 +-> DESELDD       ;OTHERWISE DESELECT DRIVE
DESELDC          := COND0         -> DRVSELDE      ;IF CC0 IS TRUE, GOTO DRIVE
                                                   ; SELECTION
                                 +-> CMDERR        ;OTHERWISE COMMAND INVALID
DESELDD          := REQ           -> DESELDD       ;WAIT FOR REQ DEACTIVATED
                                 +-> DESELDE       ;THEN GOTO DESELDE
DESELDE          := VCC           -> INIT          ;GOTO DRIVE SELECTION

DESELDA.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
DESELDB.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
DESELDC.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
DESELDD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
DESELDE.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD

;****** INVALID COMMAND ******
CMDERR           := REQ           -> CMDERR        ;WAIT FOR REQ DEACTIVATED
                                 +-> CMDERRA       ;THEN OUTPUT INVALID
                                                   ; COMMAND CODE
CMDERRA          := VCC           -> CMDERRB       ;INVALID COMMAND CODE
CMDERRB          := VCC           -> CMDERRC       ;NEGATE CODE
CMDERRC          := VCC           -> DRVSELD       ;GOTO COMMAND DECODING

CMDERR.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
CMDERRA.OUTF:= READY*CD3*CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
CMDERRB.OUTF:= READY*CD3*CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
CMDERRC.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD

;****** BOT COMMAND ******
BOTCMD           := REQ           -> BOTCMD        ;WAIT FOR REQ DEACTIVATED
                                 +-> BOTCMDA       ;THEN REWIND COMMAND CODE
BOTCMDA          := STS           -> BOTCMP        ;WHEN STS IS TRUE, NEGATE
                                                   ; OUTPUT CODE
BOTCMP           := VCC           -> DRVSELD       ;GOTO DRVSELD

BOTCMD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
BOTCMDA.OUTF:= READY*CD3*CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
BOTCMP.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD

;****** INITIALIZE COMMAND ******
INITCMD          := REQ           -> INITCMD       ;WAIT FOR REQ DEACTIVATED
                                 +-> INITCMDA      ;THEN OUTPUT REWIND COMMAND
                                                   ; CODE & ASSERT INITIALIZE
INITCMDA         := STS           -> INITCMDB      ;WHEN STS IS TRUE, NEGATE
                                                   ; COMMAND CODE
INITCMDB         := VCC           -> INITCMDC      ;ASSERT FFWD COMMAND CODE
INITCMDC         := STS           -> INITCMDD      ;WHEN STS IS TRUE, NEGATE
                                                   ; COMMAND CODE
INITCMDD         := VCC           -> INITCMDE      ;ASSERT REWIND COMMAND CODE
INITCMDE         := STS           -> INITCMP       ;WHEN STS IS TRUE, NEGATE
                                                   ; COMMAND CODE & INITIALIZE
INITCMP          := VCC           -> DRVSELD

INITCMD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
INITCMDA.OUTF:= READY*CD3*CD2*/CD1*CD0*/ERASE*INITIALIZE*DRSELD
INITCMDB.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*INITIALIZE*DRSELD
INITCMDC.OUTF:= READY*CD3*CD2*/CD1*/CD0*/ERASE*INITIALIZE*DRSELD
INITCMDD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*INITIALIZE*DRSELD
INITCMDE.OUTF:= READY*CD3*CD2*/CD1*CD0*/ERASE*INITIALIZE*DRSELD
INITCMP.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
```

```
;****** ERASE COMMAND ******
ERASECMD        := REQ          -> ERASECMD     ;WAIT FOR /REQ DEACTIVATED
                                +-> ERASECMDA   ;THEN REWIND COMMAND CODE
ERASECMDA       := STS          -> ERASECMDB    ;WHEN STS IS TRUE, NEGATE
                                                ; COMMAND CODE
ERASECMDB       := VCC          -> ERASECMDC    ;ASSERT ERASE & INITIALIZE
ERASECMDC       := STS          -> ERASECMDD    ;WHEN STS IS TRUE, NEGATE
                                                ; ERASE AND INITIALIZE
ERASECMDD       := VCC          -> ERASECMDE    ;OUTPUT REWIND COMMAND CODE
ERASECMDE       := STS          -> ERASECMDF    ;WHEN STS IS TRUE, NEGATE
                                                ; COMMAND CODE
ERASECMDF       := VCC          -> DRVSELD      ;GOTO DRVSELD

ERASECMD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
ERASECMDA.OUTF:= READY*CD3*CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
ERASECMDB.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
ERASECMDC.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*ERASE*INITIALIZE*DRSELD
ERASECMDD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
ERASECMDE.OUTF:= READY*CD3*CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
ERASECMDF.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD

;****** READ STATUS COMMAND ******
RDSTSCMD        := REQ          -> RDSTSCMD     ;WAIT FOR REQ DEACTIVATED
                                +-> RDSTSAA     ;THEN NEGATE READY
RDSTSAA         := VCC          -> RDSTSA       ;ENABLE STATUS BYTE 0
RDSTSA          := VCC          -> RDSTSB       ;ASSERT READY
RDSTSB          := REQ          -> RDSTSC       ;WHEN REQUEST IS ACTIVATED,
                                                ; NEGATE READY
RDSTSC          := REQ          -> RDSTSC       ;WAIT FOR REQUEST TO BE
                                                ; DEACTIVATED
                                +-> RDSTSD      ;THEN ENABLE STATUS BYTE 1
RDSTSD          := VCC          -> RDSTSE       ;ASSERT READY
RDSTSE          := REQ          -> RDSTSF       ;WHEN REQUEST IS ACTIVATED,
                                                ; NEGATE READY
RDSTSF          := REQ          -> RDSTSF       ;WAIT FOR REQ DEACTIVATED
                                +-> RDSTSG      ;THEN ENABLE STATUS BYTE 2
RDSTSG          := VCC          -> RDSTSH       ;ASSERT READY
RDSTSH          := REQ          -> RDSTSI       ;WHEN REQUEST IS ACTIVATED,
                                                ; NEGATE READY
RDSTSI          := REQ          -> RDSTSI       ;WAIT FOR REQ DEACTIVATED
                                +-> RDSTSJ      ;THEN ENABLE STATUS BYTE 3
RDSTSJ          := VCC          -> RDSTSK       ;ASSERT READY
RDSTSK          := REQ          -> RDSTSL       ;WHEN REQUEST IS ACTIVATED,
                                                ; NEGATE READY
RDSTSL          := REQ          -> RDSTSL       ;WAIT FOR REQ DEACTIVATED
                                +-> RDSTSM      ;THEN ENABLE STATUS BYTE 4
RDSTSM          := VCC          -> RDSTSN       ;ASSERT READY
RDSTSN          := REQ          -> RDSTSO       ;WHEN REQUEST IS ACTIVATED,
                                                ; NEGATE READY
RDSTSO          := REQ          -> RDSTSO       ;WAIT FOR REQ DEACTIVATED
                                +-> RDSTSP      ;THEN ENABLE STATUS BYTE 5
RDSTSP          := VCC          -> RDSTSQ       ;ASSERT READY
RDSTSQ          := REQ          -> RDSTSR       ;WHEN REQUEST ACTIVATED,
                                                ; NEGATE READY & COMMAND
RDSTSR          := REQ          -> RDSTSR       ;WAIT FOR REQ DEACTIVATED
                                +-> DRVSELD     ;THEN GOTO DRVSELD

RDSTSCMD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSAA.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSA.OUTF:= READY*/CD3*/CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSB.OUTF:= /READY*/CD3*/CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSC.OUTF:= READY*/CD3*/CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSD.OUTF:= READY*/CD3*/CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSE.OUTF:= /READY*/CD3*/CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSF.OUTF:= READY*/CD3*/CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSG.OUTF:= READY*/CD3*/CD2*CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSH.OUTF:= /READY*/CD3*/CD2*CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSI.OUTF:= READY*/CD3*/CD2*CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSJ.OUTF:= READY*/CD3*CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSK.OUTF:= /READY*/CD3*CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSL.OUTF:= READY*/CD3*CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSM.OUTF:= READY*/CD3*CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSN.OUTF:= /READY*/CD3*CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSO.OUTF:= READY*/CD3*CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSP.OUTF:= READY*/CD3*CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSQ.OUTF:= /READY*/CD3*CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDSTSR.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
```

```
;****** WRITE FILE MARK COMMAND ******
WFMCMD          := REQ          -> WFMCMD       ;WHEN REQUEST DEACTIVATED,
                                                ; NEGATE READY & COMMAND
                                +-> WFMCMDA     ;OTHERWISE GOTO WFMCMDA
WFMCMDA         := VCC          -> WFMCMDB      ;OUTPUT WFM COMMAND CODE
WFMCMDB         := STS          -> WFMCMDC      ;WHEN STS IS ACTIVATED,
                                                ; NEGATE COMMAND CODE
WFMCMDC         := VCC          -> DRVSELD      ;GOTO DRVSELD

WFMCMD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
WFMCMDA.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
WFMCMDB.OUTF:= READY*CD3*/CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
WFMCMDC.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD

;****** READ FILE MARK COMMAND ******
RFMCMD          := REQ          -> RFMCMD       ;WHEN REQUEST DEACTIVATED,
                                                ; NEGATE READY & COMMAND
                                +-> RFMCMDA     ;OTHERWISE GOTO RFMCMDA
RFMCMDA         := VCC          -> RFMCMDB      ;OUTPUT RFM COMMAND CODE
RFMCMDB         := STS          -> RFMCMDC      ;WHEN STS IS TRUE, NEGATE
                                                ; COMMAND CODE
RFMCMDC         := VCC          -> DRVSELD      ;GOTO DRVSELD

RFMCMD.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RFMCMDA.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RFMCMDB.OUTF:= READY*CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RFMCMDC.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD

;****** READ COMMAND ******
RDCMD           := REQ          -> RDCMD        ;WHEN REQUEST DEACTIVATED,
                                                ; NEGATE READY
                                +-> RDCMDA      ;OTHERWISE GOTO RDCMD
RDCMDA          := ONLN         -> RDCMDB       ;IF ONLINEB IS TRUE, OUTPUT
                                                ; READ COMMAND CODE
                                +-> CMDERRA     ;IF ONLINEB FALSE, OUTPUT
                                                ; INVALID COMMAND CODE
RDCMDB          := STS          -> RDCMDC       ;IF FILE MARK IS FOUND,
                                                ; NEGATE COMMAND CODE
                                +-> RDCMDD      ;OTHERWISE GOTO RDCMDD
RDCMDD          := ONLN         -> RDCMDE       ;IF ONLINE IS TRUE, GOTO
                                                ; RDCMDE
                                +-> RDCMDF      ;OTHERWISE NEGATE COMMAND
                                                ; CODE
RDCMDF          := VCC          -> RDCMDG       ;OUTPUT REWIND COMMAND CODE
RDCMDG          := STS          -> RDCMDC       ;NEGATE COMMAND CODE
RDCMDE          := REQ          -> DRVSELD      ;IF REQUEST IS TRUE, GOTO
                                                ; DRVSELD
                                +-> RDCMDB      ;OTHERWISE GOTO RDCMDB
RDCMDC          := VCC          -> DRVSELD      ;GOTO DRVSELD

RDCMD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDCMDA.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDCMDB.OUTF:= /READY*CD3*/CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDCMDC.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDCMDD.OUTF:= /READY*CD3*/CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDCMDE.OUTF:= /READY*CD3*/CD2*CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDCMDF.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
RDCMDG.OUTF:= /READY*CD3*CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD

;****** WRITE COMMAND ******
WRCMD           := REQ          -> WRCMD        ;WHEN REQUEST DEACTIVATED,
                                                ; NEGATE READY
                                +-> WRCMDA      ;OTHERWISE GOTO WRCMD
WRCMDA          := ONLN         -> WRCMDB       ;IF ONLINE IS TRUE, OUTPUT
                                                ; WRITE COMMAND CODE
                                +-> CMDERRA     ;IF ONLINE FALSE, OUTPUT
                                                ; INVALID COMMAND CODE
WRCMDB          := EW           -> WRCMDC       ;IF EWH IS DETECTED,
                                                ; NEGATE COMMAND CODE
                                +-> WRCMDE      ;OTHERWISE GOTO WRCMDE
WRCMDC          := VCC          -> WRCMDD       ;OUTPUT END OF MEDIA
                                                ; COMMAND CODE
WRCMDD          := VCC          -> DRVSELD      ;GOTO DRVSELD
WRCMDE          := REQ          -> DRVSELD      ;IF REQUEST IS TRUE, GOTO
                                                ; DRVSELD
                                +-> WRCMDF      ;OTHERWISE GOTO WRCMDF
WRCMDF          := ONLN         -> WRCMDB       ;IF ONLINEB IS TRUE, GOTO
                                                ; WRCMDB
```

```
                                    +-> WRCMDG        ;NEGATE COMMAND CODE
WRCMDG          := VCC               -> WRCMDH        ;OUTPUT WFM COMMAND CODE
WRCMDH          := STS               -> WRCMDI        ;IF FM IS ACTIVATED, GOTO
                                                      ; WRCMDI
                                    +-> WRCMDH        ;OTHERWISE GOTO WRCMDH
WRCMDI          := VCC               -> BOTCMDA       ;REWIND TAPE

WRCMD.OUTF:= READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDA.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDB.OUTF:= /READY*CD3*/CD2*CD1*CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDC.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDD.OUTF:= /READY*/CD3*CD2*CD1*CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDE.OUTF:= /READY*CD3*/CD2*CD1*CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDF.OUTF:= /READY*CD3*/CD2*CD1*CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDG.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDH.OUTF:= /READY*CD3*/CD2*/CD1*CD0*/ERASE*/INITIALIZE*DRSELD
WRCMDI.OUTF:= /READY*/CD3*/CD2*/CD1*/CD0*/ERASE*/INITIALIZE*DRSELD


;@@@@@@ CONDITIONS @@@@@@

CONDITIONS
COND0    = CC0
COND1    = CC1
COND2    = CC2
COND3    = CC3
ONLN     = ONLINE
STS      = STATUS
EW       = EWH
REQ      = REQUEST

;@@@@@@ SIMULATION INPUT @@@@@@

SIMULATION
TRACE_ON /CC0 /CC1 /CC2 /CC3 /REQUEST /ONLINE CD3 CD2 CD1 CD0 STATUS ERASE
 INITIALIZE READY DRSELD
SETF /CC3 /CC2 /CC1 /CC0 /REQUEST /ONLINE /STATUS /EWH /PRESET /CLK
SETF PRESET                        ;RESET DEVICE
CLOCKF CLK                         ;SELECT DRIVE
SETF /CC3 /CC2 /CC1 CC0
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /CC3 /CC2 /CC1 /CC0           ;INVALID COMMAND
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /CC3 CC2 /CC1 CC0             ;BOT COMMAND
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
SETF STATUS                        ;ASSERT STATUS, BOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
```

```
SETF /CC3 CC2 CC1 /CC0          ;ERASE COMMAND
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
SETF STATUS                     ; ASSERT STATUS, BOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS                     ;ASSERT STATUS, EOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS                     ;ASSERT STATUS, BOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
SETF /CC3 CC2 CC1 CC0           ;INITIALIZE COMMAND
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
SETF STATUS                     ;ASSERT STATUS, BOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS                     ;ASSERT STATUS, EOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS                     ;ASSERT STATUS, BOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
SETF /CC3 CC2 /CC1 /CC0         ;READ STATUS
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
```

```
SETF /REQUEST                    ;READ STATUS BYTE 1
CLOCKF CLK
CLOCKF CLK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST                    ;READ STATUS BYTE 1
CLOCKF CLK
CLOCKF CLK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST                    ;READ STATUS BYTE 2
CLOCKF CLK
CLOCKF CLK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST                    ;READ STATUS BYTE 3
CLOCKF CLK
CLOCKF CLK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST                    ;READ STATUS BYTE 4
CLOCKF CLK
CLOCKF CLK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST                    ;READ STATUS BYTE 5
CLOCKF CLK
CLOCKF CLK
SETF CC3 CC2 /CC1 CC0            ;READ COMMAND
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /ONLINE                     ;NEGATE ONLINEB
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
SETF CC3 CC2 /CC1 CC0            ;READ COMMAND
SETF ONLINE
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
```

```
SETF STATUS                          ;FILE MARK FOUND
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF CC3 CC2 /CC1 CC0                ;READ COMMAND
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF CC3 CC2 CC1 /CC0                ;READ FILE MARK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF CC3 CC2 /CC1 CC0                ;READ COMMAND
SETF /ONLINE
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF CC3 CC2 /CC1 /CC0               ;WRITE COMMAND
SETF ONLINE
CLOCKF CLK
SETF REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /REQUEST
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /ONLINE                         ;NEGATE ONLINEB
CLOCKF CLK
CLOCKF CLK
```

```
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS
CLOCKF CLK
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STATUS                          ;ASSERT STATUS, BOT
CLOCKF CLK
SETF /STATUS
CLOCKF CLK
CLOCKF CLK
TRACE_OFF
```

```
PLE5P16                 PLE DESIGN SPECIFICATION
12345                   KEN WON  1/27/87
COMMAND CODE DECODER
MMI, SANTA CLARA, CALIFORNIA

STS0            = /CD3*/CD2*/CD1*CD0    ; ENABLE STATUS BYTE 0
STS1            = /CD3*/CD2*CD1*/CD0    ; ENABLE STATUS BYTE 1
STS2            = /CD3*/CD2*CD1*CD0     ; ENABLE STATUS BYTE 2
STS3            = /CD3*CD2*/CD1*/CD0    ; ENABLE STATUS BYTE 3
STS4            = /CD3*CD2*/CD1*CD0     ; ENABLE STATUS BYTE 4
STS5            = /CD3*CD2*CD1*/CD0     ; ENABLE STATUS BYTE 5
EOM             = /CD3*CD2*CD1*CD0      ; END OF MEDIA
RFM             = CD3*/CD2*/CD1*/CD0    ; READ FILE MARK
WFM             = CD3*/CD2*/CD1*CD0     ; WRITE FILE MARK
READ            = CD3*/CD2*CD1*/CD0     ; READ
WRITE           = CD3*/CD2*CD1*CD0      ; WRITE
FFWD            = CD3*CD2*/CD1*/CD0     ; FAST FORWARD
RWD             = CD3*CD2*/CD1*CD0      ; REWIND
ICMD            = CD3*CD2*CD1*/CD0      ; INVALID COMMAND

FUNCTION TABLE

NC CD3 CD2 CD1 CD0 EOM ICMD RWD FFWD WRITE READ WFM RFM
 STS5 STS4 STS3 STS2 STS1 STS0

;...INPUT...../.........................OUTPUT.........................

-------------------------------------------------------------------------------
LLLLL   L L L L L L L L L L L L L L     ; NO COMMAND
LLLLH   L L L L L L L L L L L L L H     ; ENABLE STATUS BYTE 0
LLLHL   L L L L L L L L L L L L H L     ; ENABLE STATUS BYTE 1
LLLHH   L L L L L L L L L L L H L L     ; ENABLE STATUS BYTE 2
LLHLL   L L L L L L L L L L H L L L     ; ENABLE STATUS BYTE 3
LLHLH   L L L L L L L L L H L L L L     ; ENABLE STATUS BYTE 4
LLHHL   L L L L L L L L H L L L L L     ; ENABLE STATUS BYTE 5
LLHHH   H L L L L L L L L L L L L L     ; END OF MEDIA
LHLLL   L L L L L L L H L L L L L L     ; READ FILE MARK
LHLLH   L L L L L L H L L L L L L L     ; WRITE FILE MARK
LHLHL   L L L L L H L L L L L L L L     ; READ
LHLHH   L L L L H L L L L L L L L L     ; WRITE
LHHLL   L L L H L L L L L L L L L L     ; FAST FORWARD
LHHLH   L L H L L L L L L L L L L L     ; REWIND
LHHHL   L H L L L L L L L L L L L L     ; INVALID COMMAND
-------------------------------------------------------------------------------

DESCRIPTION

THIS PLE DEVICE IS ONE OF THREE DEVICES USED TO IMPLEMENT A QIC-02
COMMAND SEQUENCER.  THIS DEVICE DECODES THE OUTPUT CODE FROM THE
PROSE DEVICE TO GENERATE CONTROL SIGNALS. THE OTHER TWO DEVICES ARE
A 20L8 PAL DEVICE AND A 14R21 PROSE DEVICE.
```

IMPORTANT
NOTES

GCR (4B-5B) Encoder/Decoder

One of the more common logic functions performed on serial data is the data encode/decode function. Usually it is desirable to map (encode) the logical bit stream to a physical bit stream, adjusting for the peculiarities of the particular transmission or storage media.

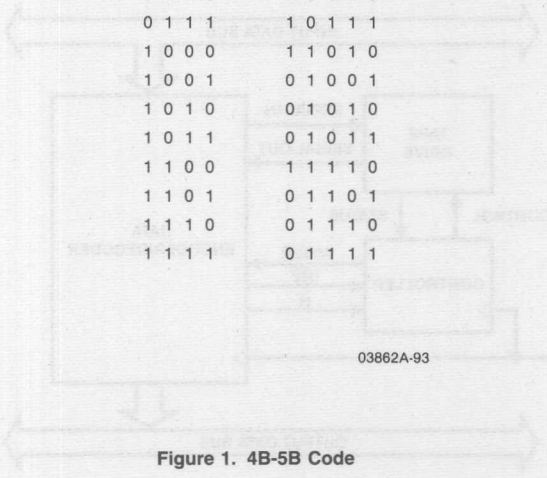
Noise, bandwidth, and reliability considerations may mean that a different data format would be desirable when data is sent along to or stored on a given media. For example, group-coded recording (GCR) formats take a given number of data bits and encode them with a larger number of bits. A 4B-5B GCR code would take 4 data bits and encode them into 16 states with 5 new bits. A particular 4B-5B code is shown in Figure 1.

This mapping allows at most two zeros to occur in succession. Also note that data combinations with more than one zero at the beginning and end of the word are excluded. This is necessary to insure that when data words are serialized, no more than two zeros occur in succession at any point in the bit stream. Finally, the data combination 11111 is reserved as a synchronization mark. In tape systems, this results in increased bit density and eases clock synchronization.

2

4B-5B Code

4-Bit Data	5-Bit Data
0 0 0 0	1 1 0 0 1
0 0 0 1	1 1 0 1 1
0 0 1 0	1 0 0 1 0
0 0 1 1	1 0 0 1 1
0 1 0 0	1 1 1 0 1
0 1 0 1	1 0 1 0 1
0 1 1 0	1 0 1 1 0
0 1 1 1	1 0 1 1 1
1 0 0 0	1 1 0 1 0
1 0 0 1	0 1 0 0 1
1 0 1 0	0 1 0 1 0
1 0 1 1	0 1 0 1 1
1 1 0 0	1 1 1 1 0
1 1 0 1	0 1 1 0 1
1 1 1 0	0 1 1 1 0
1 1 1 1	0 1 1 1 1



03862A-93

Figure 1. 4B-5B Code

troller. Parallel input data is given to the GCR E/D, converted to the 5-bit format, serialized, and written to the tape. On a read, the serial data from the tape is parallelized, converted back to the 4-bit format and output to the output data bus. Additionally, during a read, two status signals are developed. The first signal, $\overline{INV}$, indicates the presence of an invalid input, i.e., too many zeros in succession. The second status signal, $\overline{H}$, indicates the detection of the synchronization mark (11111).

The operation modes for the GCR E/D are shown in the Data-Flow Diagrams of Figure 3. The control signal and operation functions are indicated for each operation mode. In particular, the data flow between each bit of the output register is indicated schematically.

The first mode of operation of the GCR E/D is the HOLD mode. When $\overline{ENABLE}$ is HIGH, all data operations on the output register are disabled, independent of the two mode controls, M_1 and M_0. The output data is simply fed-back to the register inputs. Thus the register content is retained after the clock transition.

When the $\overline{ENABLE}$ input is LOW, the operations indicated by the M_1 and M_0 mode bits are executed on the clock transition. When M_1 and M_0 are both LOW, the SERIAL SHIFT IN mode is selected. In this mode the output register is configured as a serial shift register. The serial input is consecutively shifted into the register until all 5 bits from the tape have been stored, MSB at Y_3 and LSB at SERIAL OUT.

After the 5 bits of data have been serialized by the SERIAL SHIFT IN instruction, the 5B code must be converted to a 4B code. This is accomplished by taking the outputs of the 5 register bits and converting them to 4 bits with combinatorial logic. On the clock transition, the result is loaded into the Y register. On the same clock transition that loads the converted data into the Y register, the serial input is loaded into the serial output register. Because the serial data is being read continuously, one data bit per clock transition, the conversion must be done without missing a serial data bit.

The CONVERT PARALLEL INPUT AND LOAD operation is selected when $\overline{ENABLE}$ is LOW, M_1 is HIGH and M_0 is HIGH. This mode takes the 4 input data bits and converts them to the 5 bit representation. The result is loaded into the output register on the clock transition. The LSB of the 5B representation is loaded into the Y_3 bit of the output register and the MSB is loaded into the serial output bit. This configuration, in conjunction with the next instruction, allows the serial data to be written to the tape drive one bit per clock transition.

The final operation, SERIAL SHIFT OUT, is selected when $\overline{ENABLE}$ is LOW, M_1 is LOW and M_0 is HIGH. After the CONVERT PARALLEL INPUT AND LOAD operation is executed, the SERIAL SHIFT OUT operation outputs the converted data to the tape drive. A series of one convert operation followed by 4 shift operations will transfer a sequence of 5-bits to the tape drive, one bit per clock cycle.

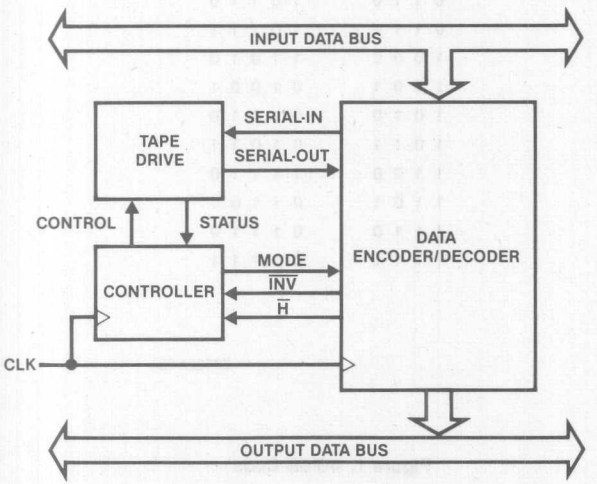


Figure 2. Typical Tape Storage System

$\overline{ENABLE}$	M_1	M_0	OPERATION	DATA-FLOW DIAGRAM
1	X	X	HOLD	
0	0	0	SERIAL SHIFT IN	
0	1	0	CONVERT SERIAL INPUT AND LOAD	
0	1	1	CONVERT PARALLEL INPUT AND LOAD	
0	0	1	SERIAL SHIFT OUT	

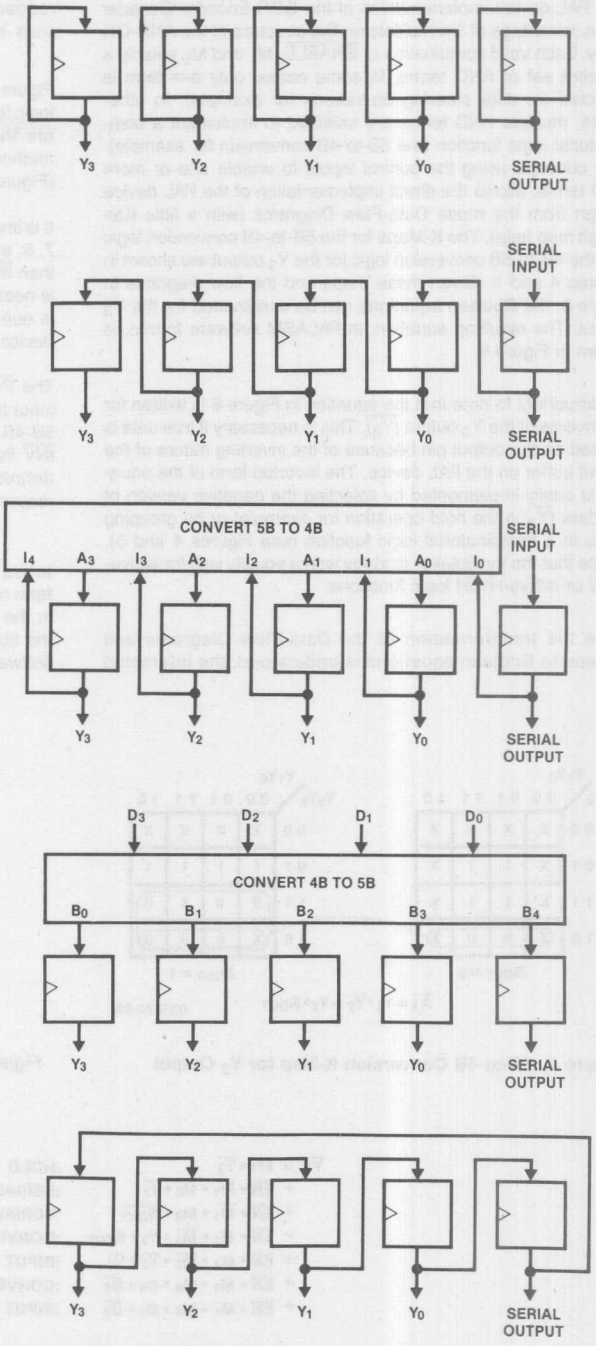


Figure 3. GCR E/D Mode Definitions

03862A-95

Design Approach (AmPAL16R6)

The PAL device implementation of the GCR Encoder/Decoder takes advantage of the multiplexer-like structure of the AND-OR array. Each valid combination of $\overline{ENABLE}$, M_1 and M_0 selects a different set of AND terms. In some cases, only one term is selected (in data steering operations for example). In other cases, multiple AND terms are selected to implement a combinatorial logic function (the 5B-to-4B conversion for example). This concept, using the control inputs to enable one or more AND terms, allows the direct implementation of the PAL device design from the mode Data-Flow Diagrams (with a little Karnaugh map help). The K-Maps for the 5B-to-4B conversion logic and the 4B-to-5B conversion logic for the Y_3 output are shown in Figures 4 and 5. Given these maps and the flow diagrams in Figure 3, the Boolean equations can be constructed for the Y_3 output. The resulting equation, in PALASM software format, is shown in Figure 6.

It is important to note that the equation in Figure 6 is written for the inverse of the Y_3 output ($\overline{Y_3}$). This is necessary if true data is desired on the output pin because of the inverting nature of the output buffer on the PAL device. The inverted form of the equation is easily implemented by selecting the negative version of the data ($\overline{Y_3}$ in the hold operation for example) or by grouping zeros in a combinatorial logic function (see Figures 4 and 5). Notice that the multiplexer strategy works equally well for active-LOW or active-HIGH logic functions.

Once the transformation of the Data-Flow Diagrams and K-Maps to Boolean equations is understood, the interested reader should be able to construct K-Maps for the other Y outputs and, in conjunction with the Data-Flow Diagrams of Figure 3, write the PALASM software equations for the resulting logic functions. This exercise will help the reader to fully appreciate the advantages of the Data-Flow Diagram/Multiplexer method of PAL design. Consult the full PALASM software listing (Figure 10) for the complete solutions.

It is important to note that the diagrams and equations in Figures 7, 8, and 9 specify the true output for invalid signal (INV), rather than $\overline{INV}$ which appears in the system diagram of Figure 2. This is necessary if the correct data is desired on the output pin and is due to the inverting nature of the output buffer on the PAL device.

The $\overline{INV}$ signal is registered and held until the clear $\overline{INV}$ flag input (CIF) is brought LOW, deactivating the flag. Only during a 5B-4B conversion operation (M_1 = HIGH, M_0 = LOW) is the $\overline{INV}$ flag activated. Figures 7 and 8 show the $\overline{INV}$ flag mode definitions and the intermediate INVALID logic equation respectively.

In this case, an active-LOW output is desired so the active-HIGH form of the $\overline{INV}$ signal is developed internally. Ones are grouped in the intermediate combinatorial logic function (INVALID) and the true version of the data is selected. The complete PALASM software equation for $\overline{INV}$ is given in Figure 9.

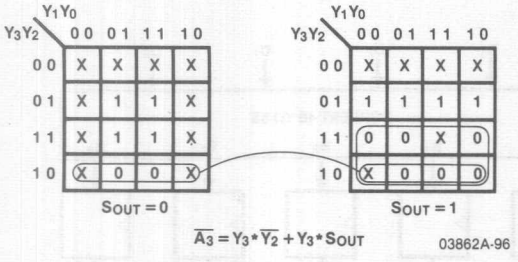


Figure 4. 5B-to-4B Conversion K-Map for Y_3 Output

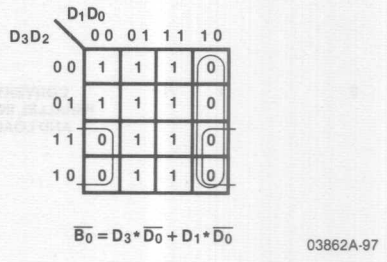


Figure 5. 4B-to-5B Conversion K-Map for Y_3 Output

```
Y3 := EN * Y3                    ;HOLD
    + EN * M1 * M0 * Y2          ;SERIAL SHIFT IN
    + EN * M1 * M0 * SOUT        ;SERIAL SHIFT OUT
    + EN * M1 * M0 * Y3 * SOUT   ;CONVERT SERIAL
    + EN * M1 * M0 * Y3 * Y2     ;INPUT AND LOAD
    + EN * M1 * M0 * D3 * D0     ;CONVERT PARALLEL
    + EN * M1 * M0 * D1 * D0     ;INPUT AND LOAD
```

03862A-98

Figure 6. PALASM Equation for Y_3

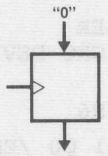
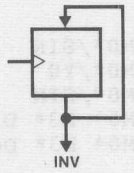
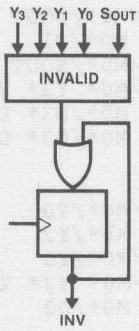
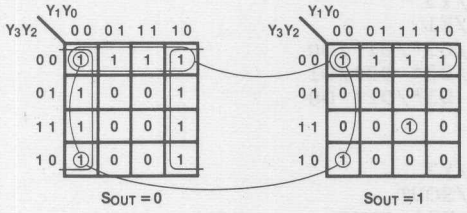
CIF	M_1	M_0	OPERATION	DATA-FLOW DIAGRAM
0	X	X	CLEAR INV FLAG	
1	0	0	HOLD INV FLAG	
	0	1		
	1	1		
1	1	0	SET INV FLAG	

Figure 7. INV Flag Mode Definitions



$$\text{INVALID} = \overline{Y_3} * \overline{Y_2} + \overline{Y_2} * \overline{Y_1} * \overline{Y_0} + \overline{Y_0} * \overline{S_{OUT}} + Y_3 * Y_2 * Y_1 * Y_0 * S_{OUT}$$

03862A-100

Figure 8. Intermediate Equation for INVALID

$$\begin{aligned}
\text{INV} := {} & \overline{CIF} * \text{INV} && + \quad \text{;HOLD INV FLAG} \\
& \overline{CIF} * M_1 * \overline{M_0} * \overline{Y_3} * \overline{Y_2} && + \quad \text{;SET INV FLAG IF INVALID IS TRUE} \\
& \overline{CIF} * M_1 * \overline{M_0} * \overline{Y_0} * \overline{S_{OUT}} && + \\
& \overline{CIF} * M_1 * \overline{M_0} * \overline{Y_2} * \overline{Y_1} * \overline{Y_0} && + \\
& \overline{CIF} * M_1 * \overline{M_0} * Y_3 * Y_2 * Y_1 * Y_0 * S_{OUT}
\end{aligned}$$

03862A-101

Figure 9. PALASM Equation for INV

```
TITLE     4B-5B ENCODER/DECODER
PATTERN   PAT003
REVISION  01
AUTHOR    WARREN MILLER
COMPANY   ADVANCED MICRO DEVICES
DATE      11/06/87

CHIP ENCO_DECO PAL16R6

 CK M1   M0  D3 D2 D1 D0 /EN   /CIF GND
/E  SIN /INV Y0 Y1 Y2 Y3  SOUT /H   VCC

EQUATIONS

/SOUT :=  EN*/SOUT                      ;HOLD
      +   /EN*/M1*/M0*/SIN              ;SERIAL SHIFT IN
      +   /EN*/M1* M0*/Y0               ;SERIAL SHIFT OUT
      +   /EN* M1*/M0*/SIN              ;CONVERT SERIAL INPUT AND LOAD
      +   /EN* M1* M0* D3* D1           ;CONVERT PARALLEL INPUT AND LOAD
      +   /EN* M1* M0* D3* D0

/Y0   :=  EN*/Y0
      +   /EN*/M1*/M0*/SOUT
      +   /EN*/M1* M0*/Y1
      +   /EN* M1*/M0*/SOUT
      +   /EN* M1*/M0* Y3* Y2*/Y0
      +   /EN* M1* M0*/D3* D1
      +   /EN* M1* M0*/D3* D2* D0

/Y1   :=  EN*/Y1
      +   /EN*/M1*/M0*/Y0
      +   /EN*/M1* M0*/Y2
      +   /EN* M1*/M0*/Y0
      +   /EN* M1*/M0* Y3* Y2
      +   /EN* M1* M0*/D2

/Y2   :=  EN*/Y2
      +   /EN*/M1*/M0*/Y1
      +   /EN*/M1* M0*/Y3
      +   /EN* M1*/M0*/Y1
      +   /EN* M1* M0*/D3*/D1*/D0
      +   /EN* M1* M0*/D3* D2*/D1
      +   /EN* M1* M0* D3*/D1* D0

/Y3   :=  EN*/Y3
      +   /EN*/M1*/M0*/Y2
      +   /EN*/M1* M0*/SOUT
      +   /EN* M1*/M0* Y3* SOUT
      +   /EN* M1*/M0* Y3*/Y2
      +   /EN* M1* M0* D3*/D0
      +   /EN* M1* M0* D1*/D0

INV   :=  /CIF* INV                     ;HOLD INV FLAG
      +   /CIF* M1*/M0*/Y3*/Y2         ;SET INV FLAG IF INVALID TRUE
      +   /CIF* M1*/M0*/Y2*/Y1*/Y0      ;
      +   /CIF* M1*/M0*/Y0*/SOUT
      +   /CIF* M1*/M0* Y3* Y2* Y1* Y0* SOUT

H     =   Y3* Y2* Y1* Y0* SOUT
; SIMULATION NOT INCLUDED
```

Figure 10. PALASM Listing

Industrial Control

Industrial control can be defined as simply as a basic state machine: process information is gathered and analyzed, and outputs feed back to the process to modify it based upon the analysis. Thus, PAL device applications are often state machines.

A very basic industrial control system is a heater thermostat. Temperature is sensed, compared to a maximum upper or lower limit, and the heater is either turned on or off. A small state machine PLD can implement this function.

Much of the circuitry in industrial control applications is analog or linear. Process information is gathered by sensors, and then the information is conditioned to eliminate noise. The conditioned signal is then converted to a digital signal, and processed by a digital processor. The control outputs are then converted to appropriate analog signals to modify the process (Figure 1).

Data Acquisition

The sensing part of the system is data acquisition. Data acquisition is important not only in industrial control applications, but also in scientific monitoring applications. Data acquisition requires the capture of analog signals and conversion to useful information that can be analyzed. This first requires conditioning of the input, to filter out noise. The signal can then be analyzed with analog circuitry, but the power of digital microprocessors makes conversion to digital logic more efficient.

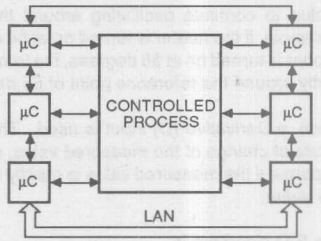
More logic is being incorporated into the acquisition circuitry. The analog circuitry for signal conditioning is a natural addition to a sensor. Analog-to-digital conversion (ADC) can even be performed in the same unit. Advanced sensors may even incorporate a microprocessor to provide immediate local analysis of the data, offloading the central processor. In these applications, PAL devices can form the glue logic for the microprocessor.

Data Analysis

Data analysis has been performed in the past by a central computer. The central computer would have the task of polling all of the sensors and analyzing the incoming data. However, in a system where thousands or millions of sensors are active, or where immediate response to process changes is required, this system is inefficient.

The type of system more commonly used today includes distributed control and processing (Figure 2). Microcomputers or programmable controllers can perform low-level processing and react with control signals immediately. Often single-board computers with added analog conversion circuitry are used.

A LAN connection between a number of intelligent sensors and processors provides a very efficient system. Simple RS-232 links connect sensors. Buses such as Multibus or VME bus are commonly used to connect the boards together. Standards such as MAP (Manufacturing Automation Protocol) are being developed for future LAN-based systems, allowing dissimilar devices to be connected in a network. Fiber-optic links will provide the data reliability over long distances in confined areas required for industrial control.



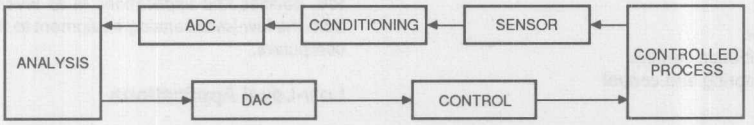
410 02

Figure 2.

In these areas, standard computer applications for PAL devices are numerous. Bus interface is another key application area for communication within distributed systems.

Control

The most critical part of the system is control. The system must respond to the analyzed information with signals that modify the process under observation. In a system such as a nuclear reactor, immediate and accurate response is the most important objective.



410 01

Figure 1.

PID Algorithm

For smooth control, the PID algorithm for analysis and control is often used. P stands for Proportional: the incoming data is compared to a reference number, and then the process is modified in proportion to the difference between the desired and measured values. For example, a heater thermostat that measures air temperature of 55 degrees, when its reference value was set at 65 degrees, would turn on the heater, and reduce the amount of heat as the temperature got closer to the reference value.

Proportional analysis alone often causes undershoot or overshoot. If the heater works quickly, it can heat the air well past the reference value. And if it works slowly, it may take a long time to heat the air back to the reference value.

For these reasons, the Integral (I) part of the algorithm is added. The integral is taken of the difference between the reference value and measured value; in effect, this measures how long the measured value has been off from the reference value. Using the integral prevents undershoot and overshoot.

However, the result can be a control output that causes the measured value to continue oscillating around the reference value. For example, if the heater is turned on at 64 degrees and an air conditioner is turned on at 66 degrees, the temperature will oscillate rapidly around the reference point of 65 degrees.

For this reason, a Derivative (D) input is used. The derivative looks at the rate of change of the measured value, and will slow down the response if the measured value is rapidly approaching the reference value.

The complete PID algorithm is

$$OUT = Kp \times ERR + Ki \times \int ERR + Kd \times dERR/dt,$$

combining present, past, and future input considerations. Selection of the proper constants allows the control response to be both rapid and smooth.

The PID algorithm requires mathematical processing capabilities, similar to those used in digital signal processing (see the section on Digital Signal Processing, page 2-283).

Control circuitry is also combining analog and digital logic. Smart power ICs can perform logic functions while receiving and sending high-power signals. These ICs can perform simple functions like overvoltage protection, cutting off a circuit if the power surges.

Applications

Applications for industrial control systems are varied, and include the following:

- Factory Automation
  - Automated distribution centers
  - Power grid monitoring and control
- Motor Control
  - Robots
  - Engine testing
  - Torque control
- Transportation
  - Air traffic control
  - Subway system control
- Automotive Electronics
- Satellite Communication
- Simulators

Performance Requirements

The key requirement of industrial control systems is the need for real-time processing. An air-traffic control system cannot be allowed to slow down if overloaded. During overloading in real systems, the information provided on display screens is reduced, instead of completely losing all information on a plane.

An interrupt handling system is more efficient than a standard polling system. Many I/O are required, with high DAC and ADC accuracy. For data analysis, a large memory and good memory management is important. Workstations with a high I/O speed are often used for central industrial controllers.

PAL Device Usage

Most of the circuitry in an industrial control application is analog or linear. Only the analyzer is primarily digital logic. The analyzer, whether local or central, uses a standard computer architecture.

Programmable logic devices, especially CMOS PLDs, are used in industrial control much as they are used in other computer applications. High speed is important in real-time control applications involving a large amount of data.

CMOS PAL Devices

Because of the harsh environment often found in factories or around motors, CMOS technology is required. Specifications for the industrial range CMOS ZPAL family are guaranteed over a wider temperature range, and a wider supply voltage range. In addition, CMOS inputs are less sensitive to noise fluctuations. The lower power consumed by all CMOS PAL devices allows a smaller power supply, important in the small areas controllers often must be placed. Also, it is easier for a battery to back up the power supply for critical operations. The reduced heat is good for enclosed areas, and allows CMOS devices to be placed closer together, even in surface-mounted devices, to save room.

The erasability of CMOS PAL devices allows changes to reference values, PID control constants, or other process control changes. Erasability also allows easy field service upgrades or repair.

PAL Device Applications

PAL devices find applications in all levels of industrial control, from the low-level sensing equipment to the mainframe analysis computers.

Low-Level Applications

Low-level data acquisition and response circuitry can be based on an individual state machine PLD. For example, stepper

motors are often used for mechanical valves. A single PAL device can be programmed to convert enable, direction, and stepping information into the appropriate signals to hold or step the motor (page 2-550).

Another example is shaft encoding (page 2-558). Shaft encoders convert the position of a rotating shaft into a digital signal. The position is sensed optically, and the digital information is encoded into position, speed, and direction signals.

More complex applications may be handled by dedicated state machine PLDs, including the PROSE device or the Am29PL141.

Medium-Range Applications

Dedicated microcontrollers can be used in medium-range applications, such as small controllers linked to a central computer. These applications require the intelligence of a microcontroller but not a complete microprocessor-based computer system. In these applications, PAL devices form the glue logic between the microcontroller and its surrounding chips.

High-End Applications

Dedicated local computers and central data processing computers are the high-end applications for PLDs. Here, microprocessors are the brains of the system and PLDs perform many of the support functions.

Microprocessor support is a key area of application (see the section titled Microprocessor-Based Systems, page 2-131). This is especially true of distributed control systems, where several microprocessors perform local processing. A standard architecture can be modified for different areas of the process by programming the PAL devices differently. PAL devices also allow the local controllers to take up less space.

Another key application area is bus interface for distributed systems (see the section titled Bus Interface, page 2-325). Either synchronous or asynchronous buses may be used.

Since industrial controllers are usually interrupt-driven for efficient response, interrupt logic forms another potential application area for PAL devices.

Digital signal processing is often performed in the analysis section of the industrial controller, usually in a central processor. The applications would not require the speed or complexity of most DSP applications, but would use similar circuitry. See the section titled Digital Signal Processing.

Stepper Motor Controllers

Functional Description

Stepper motors and linear actuators are used in a variety of applications requiring precise rotational and/or linear movement. Examples are printers, floppy disk drives, mechanical valves, etc. Stepper motors are two-phase permanent magnet motors which provide discrete angular movement every time the polarity of a winding is changed. In linear actuators the angular movement is converted to linear movement via a load screw. In essence, they are dc motors without brushes, where the user provides commutation with external logic.

In these examples, the control and drive circuitry for such a motor is implemented digitally using PAL devices.

Circuit Operation

One type of drive circuit, unipolar drive, is shown in Figure 1. Two drive sequences are given in Tables 1a and 1b. Angular rotation is achieved by saturating the transistor drivers in the sequence shown in the appropriate table (full or half step). Now, assume the circuit of Figure 1 is connected to a stepper motor designed for 7.5° steps. By following the step sequence of Table 1a (full step), the shaft will rotate 7.5° each time the state is changed. If the sequence of Table 1b is followed, a 3.75° (half-step) rotation will result for each change of state. For both step sequences, the direction can be reversed by stepping backwards through the table (step 4-3-2-1-4-etc.).

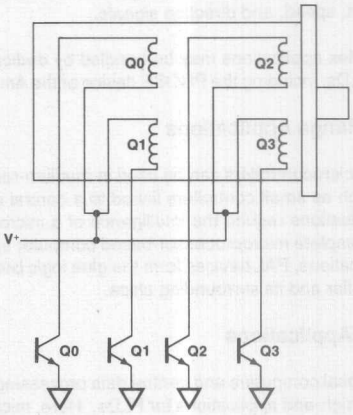


Figure 1

Table 1a

FULL STEP SEQUENCE

CLOCKWISE ROTATION ↓

STEP	Q0	Q1	Q2	Q3
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	1	0
1	1	0	1	0

↑ COUNTER-CLOCKWISE ROTATION

Table 1b

HALF STEP SEQUENCE

CLOCKWISE ROTATION ↓

STEP	Q0	Q1	Q2	Q3
1	1	0	1	0
2	1	0	0	0
3	1	0	0	1
4	0	0	0	1
5	0	1	0	1
6	0	1	0	0
7	0	1	1	0
8	0	0	1	0
1	1	0	1	0

↑ COUNTER-CLOCKWISE ROTATION

In this application, one PAL16R4 can be used to provide the logic levels required to drive two stepper motors in the full step mode. Due to the high current drive required (100-400 mA/phase), external inverting high current buffers must be used (ULN 2001 or equivalent). In the design, the following features are provided within the PAL device:

(E inputs).

- Select clockwise or counter-clockwise rotation.
- Set the motor to logic state step 1.

A block diagram/pinout is shown in Figure 2.

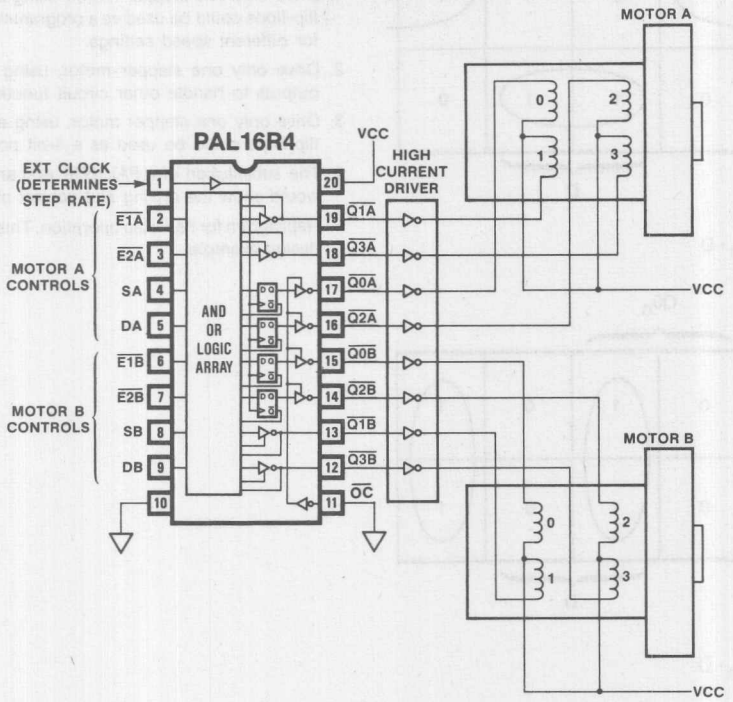


Figure 2

A function table for each motor control section is given below.

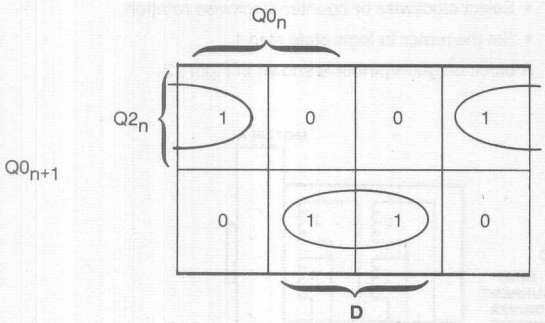
CLOCK	$\overline{E1}$	$\overline{E2}$	S	D	FUNCTION
X	1	X	X	X	Hold motor in current position
X	X	1	X	X	Hold motor in current position
↑	0	0	1	X	Set outputs to step 1 levels
↑	0	0	0	0	Step motor clockwise
↑	0	0	0	1	Step motor counter-clockwise

The full step sequence (Table 1a) can be simplified from 4 outputs to 2 outputs since Q1 = $\overline{Q0}$ and Q3 = $\overline{Q2}$. The sequence can then be expressed as follows:

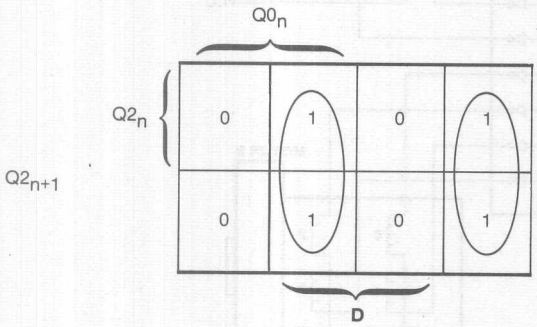
$Q0_{n+1} = 1$, $Q1_{n+1} = 0$, $Q2_{n+1} = 1$, $Q3_{n+1} = 0$
when E1 = 1 or E2 = 1:
$Q0_{n+1} = Q0_n$, $Q1_{n+1} = Q1_n$, $Q2_{n+1} = Q2_n$, $Q3_{n+1} = Q3$

	D = 0		D = 1	
STEP	Q0	Q2	Q0	Q2
1	1	1	1	1
2	1	0	0	1
3	0	0	0	0
4	0	1	1	0
1	1	1	1	1

The step sequences can be converted to equations by use of a Karnaugh map.



$$Q0_{n+1} = Q2_n \cdot \overline{D} + \overline{Q2}_n \cdot D$$



$$Q2_{n+1} = Q0_n \cdot D + \overline{Q0}_n \cdot \overline{D}$$

Factor in E1 and E2:

$$Q0_{n+1} = \overline{E1} \cdot \overline{E2} \cdot Q2_n \cdot \overline{D} + \overline{E1} \cdot \overline{E2} \cdot \overline{Q2}_n \cdot D$$
$$Q2_{n+1} = \overline{E1} \cdot \overline{E2} \cdot Q0_n \cdot \overline{D} + \overline{E1} \cdot \overline{E2} \cdot \overline{Q0}_n \cdot D$$

Express the set function as an equation:

$$Q0_{n+1} = \overline{E1} \cdot \overline{E2} \cdot S \qquad Q2_{n+1} = \overline{E1} \cdot \overline{E2} \cdot S$$

Express the hold function (when E1 or E2 = 1)

$$Q0_{n+1} = Q0_n \cdot E1 + Q0_n \cdot E2$$
$$Q2_{n+1} = Q2_n \cdot E1 + Q2_n \cdot E2$$

Combining all the above:

$$Q0_{n+1} := \overline{E1} \cdot \overline{E2} \cdot S + Q0_n \cdot E1 + Q0_n \cdot E2 + \overline{E1} \cdot \overline{E2} \cdot Q2_n \cdot \overline{D} + \overline{E1} \cdot \overline{E2} \cdot \overline{Q2}_n \cdot D$$
$$Q1_{n+1} := \overline{Q0}_{n+1}$$
$$Q2_{n+1} := \overline{E1} \cdot \overline{E2} \cdot S + Q2_n \cdot E1 + Q2_2 \cdot E2 + \overline{E1} \cdot \overline{E2} \cdot Q0_n \cdot \overline{D} + \overline{E1} \cdot \overline{E2} \cdot \overline{Q0}_n \cdot D$$
$$Q3_{n+1} := \overline{Q2}_{n+1}$$

These equations make up the first design file.

Conclusion

Although this example could be used "as is" in a stepper motor application, the programmability of PAL devices could allow for any desired modifications. Changes to the circuit might include:

1. Drive only one stepper motor, using a PAL16R6. The other flip-flops could be used as a programmable counter, allowing for different speed settings.
2. Drive only one stepper motor, using the extra inputs and outputs to handle other circuit functions.
3. Drive only one stepper motor, using a PAL16R6. The other flip-flops could be used as a 4-bit position counter.
4. The substitution of a PAL16R8, and another *inverting* buffer would allow the driving and control of four stepper motors.
5. Reprogram for half-step operation. This is done in the second design example.

```
TITLE           STEPPER MOTOR CONTROLLER
PATTERN         P7015
REVISION        1.00
AUTHOR          DAVE SACKETT
COMPANY         DEVOE COMPANY, INDIANAPOLIS, INDIANA
DATE            02/23/81

CHIP SMC PAL16R4

CLK /E1A /E2A SA DA /E1B /E2B SB DB GND /OC
/Q3B /Q1B /Q2B /Q0B /Q2A /Q0A /Q3A /Q1A VCC

EQUATIONS

Q0A :=  Q0A*/E1A                ;HOLD IF NOT E1
     +  Q0A*     /E2A           ;HOLD IF NOT E2
     +  SA * E1A* E2A           ;STEP 1 IF SET
     + /Q2A* E1A* E2A* DA       ;LOAD /Q2A IF COUNTER-CLOCKWISE
     +  Q2A* E1A* E2A*/DA       ;LOAD  Q2A IF CLOCKWISE

Q1A = /Q0A
  Q1A.TRST = VCC

Q2A :=  Q2A*/E1A                ;HOLD IF NOT E1
     +  Q2A*     /E2A           ;HOLD IF NOT E2
     +  SA * E1A* E2A           ;STEP 1 IF SET
     +  Q0A* E1A* E2A* DA       ;LOAD  Q0A IF COUNTER-CLOCKWISE
     + /Q0A* E1A* E2A*/DA       ;LOAD /Q0A IF CLOCKWISE

Q3A = /Q2A
  Q3A.TRST = VCC

Q0B :=  Q0B*/E1B                ;HOLD IF NOT E1
     +  Q0B*     /E2B           ;HOLD IF NOT E2
     +  SB * E1B* E2B           ;STEP 1 IF SET
     + /Q2B* E1B* E2B* DB       ;LOAD /Q2B IF COUNTER-CLOCKWISE
     +  Q2B* E1B* E2B*/DB       ;LOAD  Q2B IF CLOCKWISE

Q1B = /Q0B
  Q1B.TRST = VCC

Q2B :=  Q2B*/E1B                ;HOLD IF NOT E1
     +  Q2B*     /E2B           ;HOLD IF NOT E2
     +  SB * E1B* E2B           ;STEP 1 IF SET
     +  Q0B* E1B* E2B* DB       ;LOAD  Q0B IF COUNTER-CLOCKWISE
     + /Q0B* E1B* E2B*/DB       ;LOAD /Q0B IF CLOCKWISE

Q3B = /Q2B
   Q3B.TRST = VCC

SIMULATION

SETF OC E1A E2A SA E1B E2B SB
CLOCKF CLK
```

```
CHECK Q0A /Q1A Q2A /Q3A Q0B /Q1B Q2B /Q3B

SETF /E1A /E2A /E1B /E2B
CLOCKF CLK
CHECK Q0A /Q1A Q2A /Q3A Q0B /Q1B Q2B /Q3B

SETF E1A E2A /SA /DA E1B E2B /SB DB
CLOCKF CLK
CHECK Q0A /Q1A /Q2A Q3A /Q0B Q1B Q2B /Q3B

CLOCKF CLK
CHECK /Q0A Q1A /Q2A Q3A /Q0B Q1B /Q2B Q3B

CLOCKF CLK
CHECK /Q0A Q1A Q2A /Q3A Q0B /Q1B /Q2B Q3B

CLOCKF CLK
CHECK Q0A /Q1A Q2A /Q3A Q0B /Q1B Q2B /Q3B

CLOCKF CLK
CHECK Q0A /Q1A /Q2A Q3A /Q0B Q1B Q2B /Q3B

CLOCKF CLK
CHECK /Q0A Q1A /Q2A Q3A /Q0B Q1B /Q2B Q3B

SETF DA /DB
CLOCKF CLK
CHECK Q0A /Q1A /Q2A Q3A /Q0B Q1B Q2B /Q3B

SETF /E1A /E1B
CLOCKF CLK
CHECK Q0A /Q1A /Q2A Q3A /Q0B Q1B Q2B /Q3B

SETF E1A /E2A E1B /E2B
CLOCKF CLK
CHECK Q0A /Q1A /Q2A Q3A /Q0B Q1B Q2B /Q3B

SETF SA SB
CLOCKF CLK
CHECK Q0A /Q1A /Q2A Q3A /Q0B Q1B Q2B /Q3B

;DESCRIPTION
;
;THIS PAL16R4 PROVIDES THE LOGIC LEVELS REQUIRED TO DRIVE TWO
;STEPPER MOTORS IN THE FULL STEP MODE.
;
;THE FOLLOWING OPERATIONS MAY BE PERFORMED FOR EACH STEPPER
;MOTOR CONTROLLER INDIVIDUALLY:
;
; CLK   /E1   /E2   S    D    OPERATION
; ------------------------------------------------------
;  X     H     X    X    X    HOLD MOTOR IN CURRENT POSITION
;  X     X     H    X    X    HOLD MOTOR IN CURRENT POSITION
;  C     L     L    H    X    SET OUTPUTS TO STEP 1 LEVELS
;  C     L     L    L    L    STEP MOTOR CLOCKWISE
;  C     L     L    L    H    STEP MOTOR COUNTER-CLOCKWISE
; ------------------------------------------------------
```

```
PATTERN         P7094
REVISION        1.00
AUTHOR          COLI/SACKETTE
COMPANY         MONOLITHIC MEMORIES
DATE            12/07/82

CHIP DUAL_SMC PAL16R8

CLK /ENA SETA ROTA MODEA /ENB SETB ROTB MODEB GND
/OC /SW1B /SW2B /SW3B /SW4B /SW4A /SW3A /SW2A /SW1A VCC

EQUATIONS

SW1A:= SW1A                    */ENA                        ;HOLD SW1A:DISABLE
     +       /SW2A       */SW4A* ENA* ROTA* MODEA          ;HALF-STEP MTR CW
     +              SW3A        * ENA* ROTA*/MODEA          ;FULL-STEP MTR CW
     +       /SW2A*/SW3A        * ENA*/ROTA* MODEA          ;HALF-STEP MTR CCW
     +                    SW4A* ENA*/ROTA*/MODEA          ;FULL-STEP MTR CCW
     +                          ENA             * SETA    ;SET A TO STEP 1

SW2A:=        SW2A             */ENA                        ;HOLD SW2A:DISABLE
     +/SW1A       */SW3A        * ENA* ROTA* MODEA*/SETA   ;HALF-STEP MTR CW
     +                    SW4A* ENA* ROTA*/MODEA*/SETA   ;FULL-STEP MTR CW
     +/SW1A            */SW4A* ENA*/ROTA* MODEA*/SETA   ;HALF-STEP MTR CCW
     +              SW3A        * ENA*/ROTA*/MODEA*/SETA   ;FULL-STEP MTR CCW

SW3A:=              SW3A        */ENA                        ;HOLD SW3A:DISABLE
     +/SW1A            */SW4A* ENA* ROTA* MODEA          ;HALF-STEP MTR CW
     +       SW2A              * ENA* ROTA*/MODEA          ;FULL-STEP MTR CW
     +       /SW2A       */SW4A* ENA*/ROTA* MODEA          ;HALF-STEP MTR CCW
     + SW1A                    * ENA*/ROTA*/MODEA          ;FULL-STEP MTR CCW
     +                         ENA             * SETA    ;SET A TO STEP 1

SW4A:=                    SW4A*/ENA                        ;HOLD SW4A:DISABLE
     +       /SW2A*/SW3A        * ENA* ROTA* MODEA*/SETA   ;HALF-STEP MTR CW
     + SW1A                     * ENA* ROTA*/MODEA*/SETA   ;FULL-STEP MTR CW
     +/SW1A       */SW3A        * ENA*/ROTA* MODEA*/SETA   ;HALF-STEP MTR CCW
     +        SW2A              * ENA*/ROTA*/MODEA*/SETA   ;FULL-STEP MTR CCW

SW1B:= SW1B                    */ENB                        ;HOLD SW1B:DISABLE
     +       /SW2B       */SW4B* ENB* ROTB* MODEB          ;HALF-STEP MTR CW
     +              SW3B        * ENB* ROTB*/MODEB          ;FULL-STEP MTR CW
     +       /SW2B*/SW3B        * ENB*/ROTB* MODEB          ;HALF-STEP MTR CCW
     +                    SW4B* ENB*/ROTB*/MODEB          ;FULL-STEP MTR CCW
     +                          ENB             * SETB    ;SET B TO STEP 1

SW2B:=        SW2B             */ENB                        ;HOLD SW2B:DISABLE
     +/SW1B       */SW3B        * ENB* ROTB* MODEB*/SETB   ;HALF-STEP MTR CW
     +                    SW4B* ENB* ROTB*/MODEB*/SETB   ;FULL-STEP MTR CW
     +/SW1B            */SW4B* ENB*/ROTB* MODEB*/SETB   ;HALF-STEP MTR CCW
     +              SW3B        * ENB*/ROTB*/MODEB*/SETB   ;FULL-STEP MTR CCW
```

2

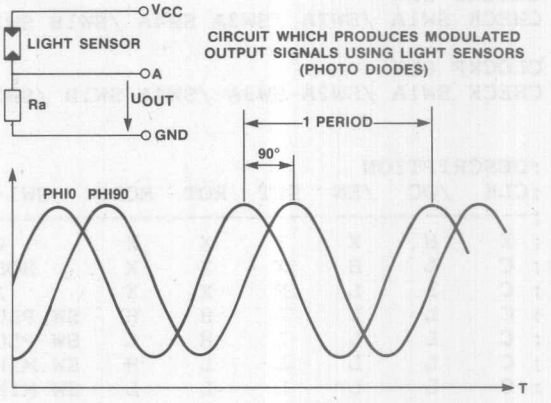
```
SW3B:=              SW3B      */ENB                          ;HOLD SW3B:DISABLE
     +/SW1B                */SW4B* ENB* ROTB* MODEB          ;HALF-STEP MTR CW
     +       SW2B                * ENB* ROTB*/MODEB          ;FULL-STEP MTR CW
     +      /SW2B         */SW4B* ENB*/ROTB* MODEB          ;HALF-STEP MTR CCW
     + SW1B                      * ENB*/ROTB*/MODEB          ;FULL-STEP MTR CCW
     +                             ENB              * SETB ;SET B TO STEP 1

SW4B:=                    SW4B*/ENB                          ;HOLD SW4B:DISABLE
     +      /SW2B*/SW3B           * ENB* ROTB* MODEB*/SETB ;HALF-STEP MTR CW
     + SW1B                      * ENB* ROTB*/MODEB*/SETB ;FULL-STEP MTR CW
     +/SW1B      */SW3B          * ENB*/ROTB* MODEB*/SETB ;HALF-STEP MTR CCW
     +       SW2B                * ENB*/ROTB*/MODEB*/SETB ;FULL-STEP MTR CCW


SIMULATION

SETF OC ENA SETA ENB SETB
CLOCKF CLK
CHECK SW1A /SW2A SW3A /SW4A SW1B /SW2B SW3B /SW4B

SETF /SETA ROTA MODEA /SETB /ROTB MODEB
CLOCKF CLK
CHECK SW1A /SW2A /SW3A /SW4A /SW1B /SW2B SW3B /SW4B

CLOCKF CLK
CHECK SW1A /SW2A /SW3A SW4A /SW1B SW2B SW3B /SW4B

SETF /ENA /ENB
CLOCKF CLK
CHECK SW1A /SW2A /SW3A SW4A /SW1B SW2B SW3B /SW4B

SETF ENA ENB
CLOCKF CLK
CHECK /SW1A /SW2A /SW3A SW4A /SW1B SW2B /SW3B /SW4B

CLOCKF CLK
CHECK /SW1A SW2A /SW3A SW4A /SW1B SW2B /SW3B SW4B

CLOCKF CLK
CHECK /SW1A SW2A /SW3A /SW4A /SW1B /SW2B /SW3B SW4B

CLOCKF CLK
CHECK /SW1A SW2A SW3A /SW4A SW1B /SW2B /SW3B SW4B

CLOCKF CLK
CHECK /SW1A /SW2A SW3A /SW4A SW1B /SW2B /SW3B /SW4B

CLOCKF CLK
CHECK SW1A /SW2A SW3A /SW4A SW1B /SW2B SW3B /SW4B

SETF /ENA /ROTA /MODEA /ENB ROTB /MODEB
CLOCKF CLK
CHECK SW1A /SW2A SW3A /SW4A SW1B /SW2B SW3B /SW4B

SETF ENA ENB
```

```
CLOCKF CLK
CHECK /SW1A SW2A SW3A /SW4A SW1B /SW2B /SW3B SW4B

CLOCKF CLK
CHECK /SW1A SW2A /SW3A SW4A /SW1B SW2B /SW3B SW4B

CLOCKF CLK
CHECK SW1A /SW2A /SW3A SW4A /SW1B SW2B SW3B /SW4B

CLOCKF CLK
CHECK SW1A /SW2A SW3A /SW4A SW1B /SW2B SW3B /SW4B


;DESCRIPTION
;CLK   /OC   /EN   SET   ROT   MODE    SW1-SW4     COMMENTS
;-------------------------------------------------------------------------
; X     H     X     X     X     X         Z         HI-Z
; C     L     H     X     X     X       HOLD        HOLD MOTOR POSITION
; C     L     L     H     X     X         1         SET MOTOR POSITION TO STEP 1
; C     L     L     L     H     H    SW PLUS  1     HALF-STEP MOTOR CLOCKWISE
; C     L     L     L     H     L    SW PLUS  2     FULL-STEP MOTOR CLOCKWISE
; C     L     L     L     L     H    SW MINUS 1     HALF-STEP MOTOR COUNTERCLOCKWI
; C     L     L     L     L     L    SW MINUS 2     FULL-STEP MOTOR COUNTERCLOCKWI
;-------------------------------------------------------------------------
```

Shaft Encoders

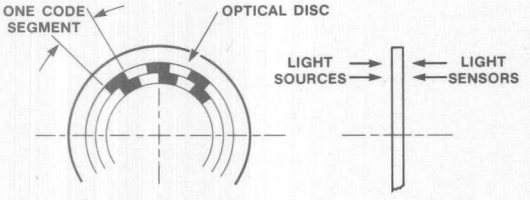
The trend away from analog systems to digital numerical control systems focuses new attention on instruments that convert the analog output of position-sensing devices into digital format.

The recording of the position of moveable parts of a machine requires high accuracy and noise immunity which can be attained through shaft encoder circuits of the type used in speed controllers and optical position-sensing devices.



Principles of Shaft Encoding

Most commonly used are opto-electrical encoders. The advantages of this type of circuit are direct shaft-to-digital encoding, a minimum number of power supplies, low power requirements, low cost and high speed.



Optical encoders measure shaft rotation by detecting the light which passes through a rotating code disc and a fixed slit.

It consists of a number of light sources and sensors whose paths are interrupted by a disc that has transparent and opaque areas.

By using various concentric patterns, the shaft position can be defined. Light shines through the disc onto sensors. A sensor turns on when the light passes through its corresponding transparent part of the disc giving an output that depends on the opacity of the segment. The combined output of all the segments is digital information representing the disc and the shaft position.

— An *absolute encoder* has a number of concentric tracks on the code disc, which provide parallel readout of shaft angle without counting pulses. The code patterns on the disc are a kind of storage.

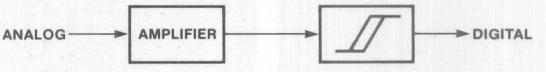
  Thus, the readout is also present after a power interruption.

— *Incremental encoders,* which will be described here, have a single code track on the disc. Angular position is determined by counting pulses produced by the modulated light falling onto the photodetectors.

  Direction sensing is obtained by the use of quadrature signals which are provided by appropriate phasing of the paths.

Thus, if power is interrupted an incremental encoder must have its zero position reestablished.

The two signals from the light sensors have to be amplified and converted into digital format using Schmitt triggers.

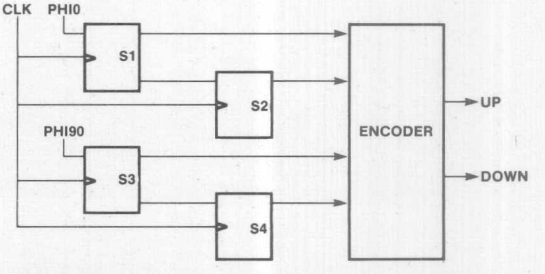


The two signals are out of phase by a quarter of a period. The direction of movement can be determined by which signal leads or lags the other.

Out of this phase relationship the shaft encoder produces information about direction and speed for a following up/down-counter.

To avoid random discrepancies during switching operations shaft encoders should be built using synchronous clocked circuitry.

The diagram below shows a typical circuit diagram for shaft encoding.



The phase relationship of the two signals PHI0 and PHI90 is determined by the four registers, and their outputs are encoded.

The underlying principle of the above circuit is the time delay between the clock pulses for the two signals PHI0 and PHI90.

S1 = PHI0 at CLK S2 = PHI0 at CLK + 1
S1 = PHI90 at CLK S4 = PHI90 at CLK + 1

A transition in PHI0 or PHI90 from L to H or vice versa causes a change in the data stored in the registers, which is encoded for direction sensing.

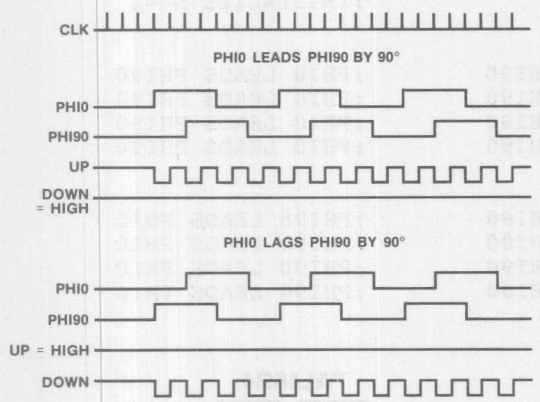
The logic equations for the encoder circuitry are given below

PHI0 leads PHI90:

```
S1 * S2 * S3 * /S4
+/S1 * /S2 * /S3 * S4
+ S1 * /S2 * /S3 * /S4
+/S1 * S2 * S3 * S4
```

PHI0 lags PHI90:

```
/S1 * /S2 * S3 * /S4
+ S1 * S2 * /S3 * S4
+ S1 * /S2 * S3 * S4
+/S1 * S2 * /S3 * /S4
```



When the shaft rotates in a clockwise direction, the input PHI0 *leads* the input PHI90 by 90° and the logic will generate pulses only at the UP-output.

On the other hand, when the shaft's rotation is counterclockwise the input PHI0 *lags* the input PHI90 by 90°. In this case, four pulses per square wave are presented at the DOWN-output. Note that both the UP and DOWN outputs of the counter are normally held high.

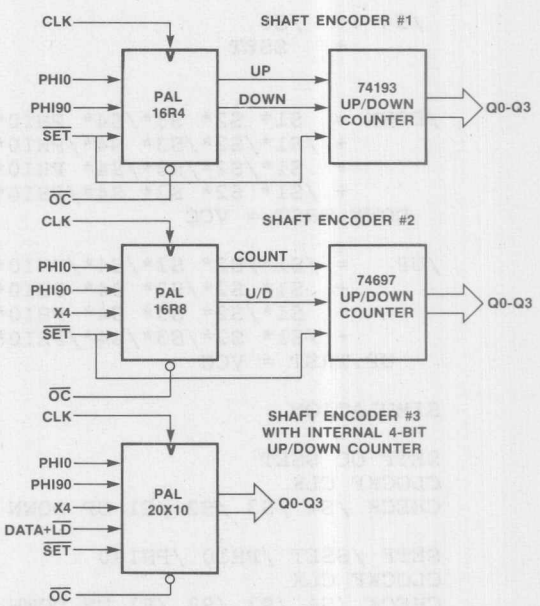
To ensure that no shaft encoder transition is missed, the clock frequency should be at least 8xNxS, where N is the number of pulses produced by the encoder for each shaft revolution and S is the maximum speed in revolutions per second to be expected.

Most commonly CLK frequencies in the range of 1 MHz are used for optimum circuit operation.

Synchronous two-channel shaft encoders as described above are relatively insensitive to encoder phase errors. They use no temperamental Mono-Flops and can detect any illegal transition states generated by the circuit.

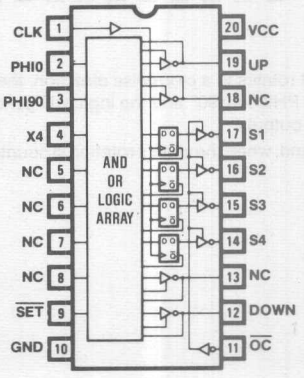
Interference on the input lines is ignored if it occurs between two clock cycles. Random noise on both input lines results in one UP count for each DOWN count so that in the end the digital information remains unchanged. Hence synchronous shaft encoders are extremely useful in electrically noisy environments.

The PAL Device Applications are examples of such circuits using a single PAL device each.



```
TITLE           SHAFT ENCODER No. 1
PATTERN         P7097
REVISION        1.00
AUTHOR          WILLY VOLDAN
COMPANY         MONOLITHIC MEMORIES
DATE            09/09/82

CHIP SE1 PAL16R4

CLK PHI0 PHI90 X4 NC NC NC NC /SSET GND
/OC DOWN NC S4 S3 S2 S1 NC UP VCC

EQUATIONS

 /S1 :=  /PHI0                              ;CHECK FOR PHI0
      +   SSET                              ;INITIALIZE S1=L

 /S2 :=  /S1                                ;CHECK FOR S1
      +   SSET                              ;INITIALIZE S2=L

 /S3 :=  /PHI90                             ;CHECK FOR PHI90
      +   SSET                              ;INITIALIZE S3=L

 /S4 :=  /S3                                ;CHECK FOR S3
      +   SSET                              ;INITIALIZE S4=L


/DOWN =   S1* S2* S3*/S4* PHI0* PHI90       ;PHI0 LEADS PHI90
      + /S1*/S2*/S3* S4*/PHI0*/PHI90        ;PHI0 LEADS PHI90
      +  S1*/S2*/S3*/S4* PHI0*/PHI90        ;PHI0 LEADS PHI90
      + /S1* S2* S3* S4*/PHI0* PHI90        ;PHI0 LEADS PHI90
  DOWN.TRST = VCC

/UP    =  /S1*/S2* S3*/S4*/PHI0* PHI90      ;PHI90 LEADS PHI0
      +   S1* S2*/S3* S4* PHI0*/PHI90       ;PHI90 LEADS PHI0
      +   S1*/S2* S3* S4* PHI0* PHI90       ;PHI90 LEADS PHI0
      + /S1* S2*/S3*/S4*/PHI0*/PHI90        ;PHI90 LEADS PHI0
   UP.TRST = VCC

SIMULATION

SETF OC SSET
CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF /SSET /PHI0 /PHI90
CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF PHI90
CLOCKF CLK
```

PAL16R4



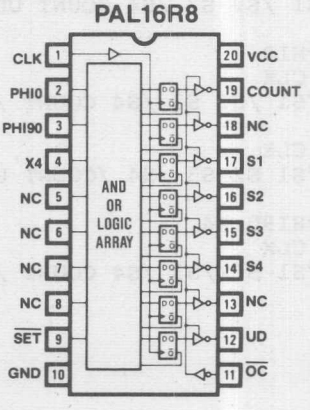
```
CHECK /S4 S3 /S2 /S1 /UP DOWN

CLOCKF CLK
CHECK S4 S3 /S2 /S1 UP DOWN

SETF PHI0
CLOCKF CLK
CHECK S4 S3 /S2 S1 /UP DOWN

CLOCKF CLK
CHECK S4 S3 S2 S1 UP DOWN

SETF /PHI90
CLOCKF CLK
CHECK S4 /S3 S2 S1 /UP DOWN

CLOCKF CLK
CHECK /S4 /S3 S2 S1 UP DOWN

SETF /PHI0
CLOCKF CLK
CHECK /S4 /S3 S2 /S1 /UP DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF PHI90
CLOCKF CLK
CHECK /S4 S3 /S2 /S1 /UP DOWN

CLOCKF CLK
CHECK S4 S3 /S2 /S1 UP DOWN

SETF PHI0
CLOCKF CLK
CHECK S4 S3 /S2 S1 /UP DOWN

CLOCKF CLK
CHECK S4 S3 S2 S1 UP DOWN

SETF /PHI90
CLOCKF CLK
CHECK S4 /S3 S2 S1 /UP DOWN

CLOCKF CLK
CHECK /S4 /S3 S2 S1 UP DOWN

SETF /PHI0
CLOCKF CLK
CHECK /S4 /S3 S2 /S1 /UP DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN
```

```
SETF PHI90
CLOCKF CLK
CHECK /S4 S3 /S2 /S1 /UP DOWN

CLOCKF CLK
CHECK S4 S3 /S2 /S1 UP DOWN

SETF PHI0
CLOCKF CLK
CHECK S4 S3 /S2 S1 /UP DOWN

CLOCKF CLK
CHECK S4 S3 S2 S1 UP DOWN

SETF /PHI90
CLOCKF CLK
CHECK S4 /S3 S2 S1 /UP DOWN

CLOCKF CLK
CHECK /S4 /S3 S2 S1 UP DOWN

SETF /PHI0
CLOCKF CLK
CHECK /S4 /S3 S2 /S1 /UP DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF SSET
CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF /SSET
CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF PHI0
CLOCKF CLK
CHECK /S4 /S3 /S2 S1 UP /DOWN

CLOCKF CLK
CHECK /S4 /S3 S2 S1 UP DOWN

SETF PHI90
CLOCKF CLK
CHECK /S4 S3 S2 S1 UP /DOWN

CLOCKF CLK
CHECK S4 S3 S2 S1 UP DOWN

SETF /PHI0
```

```
CLOCKF CLK
CHECK S4 S3 S2 /S1 UP /DOWN

CLOCKF CLK
CHECK S4 S3 /S2 /S1 UP DOWN

SETF /PHI90
CLOCKF CLK
CHECK S4 /S3 /S2 /S1 UP /DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF PHI0
CLOCKF CLK
CHECK /S4 /S3 /S2 S1 UP /DOWN

CLOCKF CLK
CHECK /S4 /S3 S2 S1 UP DOWN

SETF PHI90
CLOCKF CLK
CHECK /S4 S3 S2 S1 UP /DOWN

CLOCKF CLK
CHECK S4 S3 S2 S1 UP DOWN

SETF /PHI0
CLOCKF CLK
CHECK S4 S3 S2 /S1 UP /DOWN

CLOCKF CLK
CHECK S4 S3 /S2 /S1 UP DOWN

SETF /PHI90
CLOCKF CLK
CHECK S4 /S3 /S2 /S1 UP /DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

SETF PHI0
CLOCKF CLK
CHECK /S4 /S3 /S2 S1 UP /DOWN

CLOCKF CLK
CHECK /S4 /S3 S2 S1 UP DOWN

SETF PHI90
CLOCKF CLK
CHECK /S4 S3 S2 S1 UP /DOWN

CLOCKF CLK
CHECK S4 S3 S2 S1 UP DOWN
```

```
SETF /PHI0
CLOCKF CLK
CHECK S4 S3 S2 /S1 UP /DOWN

CLOCKF CLK
CHECK S4 S3 /S2 /S1 UP DOWN

SETF /PHI90
CLOCKF CLK
CHECK S4 /S3 /S2 /S1 UP /DOWN

CLOCKF CLK
CHECK /S4 /S3 /S2 /S1 UP DOWN

;DESCRIPTION:
;
;THIS PAL16R4 IMPLEMENTS A TWO CHANNEL SHAFT ENCODER OF THE
;TYPE USED IN SPEED CONTROLLERS AND OPTICAL DEVICES.
;
;BOTH THE "UP" AND "DOWN" OUTPUTS OF THE PAL DEVICE ARE
;NORMALLY HIGH.
;
;WHEN THE SIGNAL AT THE "PHI0" INPUT LEADS THE SIGNAL AT THE
;"PHI90" INPUT, THE "DOWN" OUTPUT ALTERNATES BETWEEN HIGH AND
;LOW LEVELS AT HALF THE "CLK" FREQUENCY RATE.  ALSO, WHEN THE
;SIGNAL AT THE "PHI0" INPUT LAGS THE SIGNAL AT THE "PHI90"
;INPUT, THE "UP" OUTPUT ALTERNATES BETWEEN HIGH AND LOW
;LEVELS AT HALF THE "CLK" FREQUENCY RATE.
;
;THE SHAFT ENCODER FEATURES THE CONFIGURATION AND OUTPUT
;POLARITY TO DRIVE AN 74S193 TYPE UP/DOWN COUNTER.
;
;THIS DESIGN WITH GLITCH-FREE OUTPUTS WILL BE EXTREMELY
;USEFUL IN ELECTRICALLY NOISY ENVIRONMENTS.  THE PINOUT IS
;GIVEN AS A FIRST PROPOSAL AND CAN BE CHANGED ACCORDING TO
;THE PC BOARD LAYOUT.
```

```
TITLE           SHAFT ENCODER No. 2
PATTERN         P7098
REVISION        1.00
AUTHOR          WILLY VOLDAN
COMPANY         MONOLITHIC MEMORIES
DATE            09/09/82

CHIP SE2 PAL16R8

CLK PHI0 PHI90 X4 NC NC NC NC /SSET GND
/OC UD NC S4 S3 S2 S1 NC COUNT VCC

EQUATIONS

/S1   := /PHI0                          ;CHECK FOR PHI0
       +  SSET                          ;INITIALIZE S1=L

/S3   := /PHI90                         ;CHECK FOR PHI90
       +  SSET                          ;INITIALIZE S3=L

/S2   :=  S1                            ;CHECK FOR /S1
       +  SSET                          ;INITIALIZE S2=L

/S4   :=  S3                            ;CHECK FOR /S3
       +  SSET                          ;INITIALIZE S4=L

/COUNT :=  S1* S2*/S3* S4               ;THIS OUTPUT ALTERNATES
        + /S1*/S2* S3*/S4               ;BETWEEN HIGH AND LOW WITH
        + /S1* S2*/S3*/S4* X4           ;HALF OR QUARTER THE
        +  S1*/S2* S3* S4* X4           ;CLK FREQUENCY
        +  S1* S2* S3*/S4
        + /S1*/S2*/S3* S4
        + /S1* S2* S3* S4* X4
        +  S1*/S2*/S3*/S4* X4

/UD   := /S1* S2*/S3* S4                ;THIS OUTPUT DETERMINES
        + /S1* S2* S3* S4               ;IF SIGNAL PHI0 LEADS
        + /S1* S2* S3*/S4               ;OR LAGS SIGNAL PHI90
        +  S1* S2* S3*/S4
        +  S1*/S2* S3*/S4
        +  S1*/S2*/S3*/S4
        +  S1*/S2*/S3* S4
        + /S1*/S2*/S3* S4


SIMULATION

SETF OC SSET
CLOCKF CLK
CHECK /S1 /S2 /S3 /S4

SETF /SSET /PHI0 /PHI90 /X4
CLOCKF CLK
CHECK /S1 S2 /S3 S4 COUNT UD
```



```
SETF PHI0
CLOCKF CLK
CHECK S1 S2 /S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 /S3 S4 /COUNT UD

SETF PHI90
CLOCKF CLK
CHECK S1 /S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 COUNT UD

SETF /PHI0
CLOCKF CLK
CHECK /S1 /S2 S3 /S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 S3 /S4 /COUNT UD

SETF /PHI90
CLOCKF CLK
CHECK /S1 S2 /S3 /S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 /S3 S4 COUNT UD

SETF PHI0
CLOCKF CLK
CHECK S1 S2 /S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 /S3 S4 /COUNT UD

SETF PHI90
CLOCKF CLK
CHECK S1 /S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 COUNT UD

SETF /PHI0
CLOCKF CLK
CHECK /S1 /S2 S3 /S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 S3 /S4 /COUNT UD

SETF /PHI90 X4
CLOCKF CLK
CHECK /S1 S2 /S3 /S4 COUNT /UD
```

```
CLOCKF CLK
CHECK /S1 S2 /S3 S4 /COUNT UD

SETF PHI0
CLOCKF CLK
CHECK S1 S2 /S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 /S3 S4 /COUNT UD

SETF PHI90
CLOCKF CLK
CHECK S1 /S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 /COUNT UD

SETF /PHI0
CLOCKF CLK
CHECK /S1 /S2 S3 /S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 S3 /S4 /COUNT UD

SETF /PHI90
CLOCKF CLK
CHECK /S1 S2 /S3 /S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 /S3 S4 /COUNT UD

SETF PHI0
CLOCKF CLK
CHECK S1 S2 /S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 /S3 S4 /COUNT UD

SETF PHI90
CLOCKF CLK
CHECK S1 /S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 /COUNT UD

SETF SSET
CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 COUNT /UD

SETF /SSET /PHI0 /PHI90 /X4
CLOCKF CLK
CHECK /S1 S2 /S3 S4 COUNT UD

SETF PHI90
```

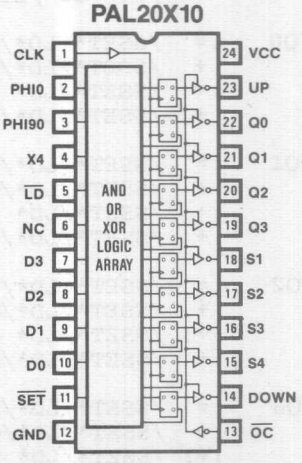
```
CLOCKF CLK
CHECK /S1 S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 S3 /S4 COUNT /UD

SETF PHI0
CLOCKF CLK
CHECK S1 S2 S3 /S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 /COUNT /UD

SETF /PHI90
CLOCKF CLK
CHECK S1 /S2 /S3 /S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 /S3 S4 COUNT /UD

SETF /PHI0
CLOCKF CLK
CHECK /S1 /S2 /S3 S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 /S3 S4 /COUNT /UD

SETF PHI90
CLOCKF CLK
CHECK /S1 S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 S3 /S4 COUNT /UD

SETF PHI0
CLOCKF CLK
CHECK S1 S2 S3 /S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 /COUNT /UD

SETF /PHI90
CLOCKF CLK
CHECK S1 /S2 /S3 /S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 /S3 S4 COUNT /UD

SETF /PHI0 X4
CLOCKF CLK
CHECK /S1 /S2 /S3 S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 /S3 S4 /COUNT /UD
```

```
SETF PHI90
CLOCKF CLK
CHECK /S1 S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 S3 /S4 /COUNT /UD

SETF PHI0
CLOCKF CLK
CHECK S1 S2 S3 /S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 /COUNT /UD

SETF /PHI90
CLOCKF CLK
CHECK S1 /S2 /S3 /S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 /S3 S4 /COUNT /UD

SETF /PHI0
CLOCKF CLK
CHECK /S1 /S2 /S3 S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 /S3 S4 /COUNT /UD

SETF PHI90
CLOCKF CLK
CHECK /S1 S2 S3 S4 COUNT /UD

CLOCKF CLK
CHECK /S1 S2 S3 /S4 /COUNT /UD

SETF PHI0
CLOCKF CLK
CHECK S1 S2 S3 /S4 COUNT /UD

CLOCKF CLK
CHECK S1 /S2 S3 /S4 /COUNT /UD

;DESCRIPTION
;
;THIS PAL16R8 IMPLEMENTS A TWO CHANNEL SHAFT ENCODER OF THE
;TYPE USED IN SPEED CONTROLLERS AND OPTICAL DEVICES.
;
;THE "COUNT" OUTPUT OF THE PAL DEVICE IS NORMALLY HIGH.  DURING
;SHAFT ENCODING THIS OUTPUT ALTERNATES BETWEEN HIGH AND LOW.
;
;INPUT "X4" SELECTS BETWEEN HALF (X4=H) OR QUARTER (X4=L) CLK
;FREQUENCY OF THE "COUNTER" OUTPUT.
;
```

```
;OUTPUT "UD" DETERMINES WHETHER SIGNAL PHI0 LEADS (UD=H) OR
;LAGS (UD=L) SIGNAL PHI90.
;
;THE SHAFT ENCODER FEATURES THE CONFIGURATION AND OUTPUT
;POLARITY TO DRIVE A 74S697 TYPE UP/DOWN COUNTER.
;
;THIS DESIGN WITH GLITCH-FREE OUTPUTS WILL BE EXTREMELY USEFUL
;IN ELECTRICALLY NOISY ENVIRONMENTS.  THE PINNING IS GIVEN AS
;A FIRST PROPOSAL AND CAN BE CHANGED ACCORDING TO THE PC-BOARD
;LAYOUT.
```

```
TITLE SHAFT ENCODER No. 3 (WITH INTERNAL 4-BIT UP/DOWN COUNTER)
PATTERN          P7099
REVISION         1.00
AUTHOR           WILLY VOLDAN
COMPANY          MONOLITHIC MEMORIES
DATE             09/09/82

CHIP SE3 PAL20X10

CLK PHI0 PHI90 X4 /LD NC D3 D2 D1 D0 /SSET
GND /OC DOWN S4 S3 S2 S1 Q3 Q2 Q1 Q0 UP VCC

EQUATIONS

/S1  :=  /PHI0                                  ;CHECK FOR PHI0
     +   SSET                                   ;INITIALIZE S1=L

/S2  :=  /S1                                    ;CHECK FOR S1
     +   SSET                                   ;INITIALIZE S2=L

/S3  :=  /PHI90                                 ;CHECK FOR PHI90
     +   SSET                                   ;INITIALIZE S3=L

/S4  :=  /S3                                    ;CHECK FOR S3
     +   SSET                                   ;INITIALIZE S4=L

/DOWN :=   S1* S2* S3*/S4* PHI0* PHI90* X4      ;PHI0 LEADS PHI90:COUNT=F/2
      +  /S1*/S2*/S3* S4*/PHI0*/PHI90* X4       ;PHI0 LEADS PHI90:COUNT=F/2
      :+:  S1*/S2*/S3*/S4* PHI0*/PHI90          ;PHI0 LEADS PHI90:COUNT=F/4
      +  /S1* S2* S3* S4*/PHI0* PHI90            ;PHI0 LEADS PHI90:COUNT=F/4

/UP   :=  /S1*/S2* S3*/S4*/PHI0* PHI90          ;PHI90 LEADS PHI0:COUNT=F/4
      +    S1* S2*/S3* S4* PHI0*/PHI90          ;PHI90 LEADS PHI0:COUNT=F/4
      :+:  S1*/S2* S3* S4* PHI0* PHI90* X4       ;PHI90 LEADS PHI0:COUNT=F/2
      +  /S1* S2*/S3*/S4*/PHI0*/PHI90* X4       ;PHI90 LEADS PHI0:COUNT=F/2

/Q0   :=  /SSET* LD*/D0                         ;LOAD D0 (LSB)
      +   /SSET*/LD*/Q0                         ;HOLD Q0
      :+: /SSET*/LD* UP*/DOWN                   ;DECREMENT
      +   /SSET*/LD*/UP* DOWN                   ;INCREMENT

/Q1   :=  /SSET* LD*/D1                         ;LOAD D1
      +   /SSET*/LD*/Q1                         ;HOLD Q1
      :+: /SSET*/LD* UP*/DOWN*/Q0               ;DECREMENT
      +   /SSET*/LD*/UP* DOWN* Q0               ;INCREMENT

/Q2   :=  /SSET* LD*/D2                         ;LOAD D2
      +   /SSET*/LD*/Q2                         ;HOLD Q2
      :+: /SSET*/LD* UP*/DOWN*/Q0*/Q1           ;DECREMENT
      +   /SSET*/LD*/UP* DOWN* Q0* Q1           ;INCREMENT

/Q3   :=  /SSET* LD*/D3                         ;LOAD D3 (MSB)
      +   /SSET*/LD*/Q3                         ;HOLD Q3
      :+: /SSET*/LD* UP*/DOWN*/Q0*/Q1*/Q2       ;DECREMENT
```

```
      +  /SSET*/LD*/UP* DOWN* Q0* Q1* Q2    ;INCREMENT


SIMULATION

SETF OC SSET /PHI0 /PHI90
CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 Q3 Q2 Q1 Q0

SETF /SSET LD D3 /D2 D1 /D0
CLOCKF CLK
CHECK Q3 /Q2 Q1 /Q0

SETF /LD /X4
CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 UP DOWN Q3 /Q2 Q1 /Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 /S3 /S4 UP DOWN Q3 /Q2 Q1 /Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 UP /DOWN Q3 /Q2 Q1 /Q0

SETF PHI90
CLOCKF CLK
CHECK S1 S2 S3 /S4 UP DOWN Q3 /Q2 /Q1 Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 UP DOWN Q3 /Q2 /Q1 Q0

SETF /PHI0
CLOCKF CLK
CHECK /S1 S2 S3 S4 UP DOWN Q3 /Q2 /Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 UP /DOWN Q3 /Q2 /Q1 Q0

SETF /PHI90
CLOCKF CLK
CHECK /S1 /S2 /S3 S4 UP DOWN Q3 /Q2 /Q1 /Q0

CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 UP DOWN Q3 /Q2 /Q1 /Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 /S3 /S4 UP DOWN Q3 /Q2 /Q1 /Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 UP /DOWN Q3 /Q2 /Q1 /Q0

SETF PHI90
CLOCKF CLK
```



```
CHECK S1 S2 S3 /S4 UP DOWN /Q3 Q2 Q1 Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 UP DOWN /Q3 Q2 Q1 Q0

SETF /PHI0
CLOCKF CLK
CHECK /S1 S2 S3 S4 UP DOWN /Q3 Q2 Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 UP /DOWN /Q3 Q2 Q1 Q0

SETF X4 /PHI90
CLOCKF CLK
CHECK /S1 /S2 /S3 S4 UP DOWN /Q3 Q2 Q1 /Q0

CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 UP /DOWN /Q3 Q2 Q1 /Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 /S3 /S4 UP DOWN /Q3 Q2 /Q1 Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 UP /DOWN /Q3 Q2 /Q1 Q0

SETF PHI90
CLOCKF CLK
CHECK S1 S2 S3 /S4 UP DOWN /Q3 Q2 /Q1 /Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 UP /DOWN /Q3 Q2 /Q1 /Q0

SETF /PHI0
CLOCKF CLK
CHECK /S1 S2 S3 S4 UP DOWN /Q3 /Q2 Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 UP /DOWN /Q3 /Q2 Q1 Q0

SETF /PHI90
CLOCKF CLK
CHECK /S1 /S2 /S3 S4 UP DOWN /Q3 /Q2 Q1 /Q0

CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 UP /DOWN /Q3 /Q2 Q1 /Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 /S3 /S4 UP DOWN /Q3 /Q2 /Q1 Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 UP /DOWN /Q3 /Q2 /Q1 Q0
```

```
SETF PHI90
CLOCKF CLK
CHECK S1 S2 S3 /S4 UP DOWN /Q3 /Q2 /Q1 /Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 UP /DOWN /Q3 /Q2 /Q1 /Q0

SETF /PHI0
CLOCKF CLK
CHECK /S1 S2 S3 S4 UP DOWN Q3 Q2 Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 UP /DOWN Q3 Q2 Q1 Q0

SETF /PHI90
CLOCKF CLK
CHECK /S1 /S2 /S3 S4 UP DOWN Q3 Q2 Q1 /Q0

SETF SSET /PHI0 /PHI90
CLOCKF CLK

SETF /SSET LD /D3 D2 /D1 D0
CLOCKF CLK
CHECK /Q3 Q2 /Q1 Q0

SETF /LD /X4
CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 UP DOWN /Q3 Q2 /Q1 Q0

SETF PHI90
CLOCKF CLK
CHECK /S1 /S2 S3 /S4 UP DOWN /Q3 Q2 /Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 /UP DOWN /Q3 Q2 /Q1 Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 S3 S4 UP DOWN /Q3 Q2 Q1 /Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 UP DOWN /Q3 Q2 Q1 /Q0

SETF /PHI90
CLOCKF CLK
CHECK S1 S2 /S3 S4 UP DOWN /Q3 Q2 Q1 /Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 /UP DOWN /Q3 Q2 Q1 /Q0

SETF /PHI0
CLOCKF CLK
CHECK /S1 S2 /S3 /S4 UP DOWN /Q3 Q2 Q1 Q0
```

```
CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 UP DOWN /Q3 Q2 Q1 Q0

SETF PHI90
CLOCKF CLK
CHECK /S1 /S2 S3 /S4 UP DOWN /Q3 Q2 Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 /UP DOWN /Q3 Q2 Q1 Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 S3 S4 UP DOWN Q3 /Q2 /Q1 /Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 UP DOWN Q3 /Q2 /Q1 /Q0

SETF /PHI90
CLOCKF CLK
CHECK S1 S2 /S3 S4 UP DOWN Q3 /Q2 /Q1 /Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 /UP DOWN Q3 /Q2 /Q1 /Q0

SETF X4 /PHI0
CLOCKF CLK
CHECK /S1 S2 /S3 /S4 UP DOWN Q3 /Q2 /Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 /UP DOWN Q3 /Q2 /Q1 Q0

SETF PHI90
CLOCKF CLK
CHECK /S1 /S2 S3 /S4 UP DOWN Q3 /Q2 Q1 /Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 /UP DOWN Q3 /Q2 Q1 /Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 S3 S4 UP DOWN Q3 /Q2 Q1 Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 /UP DOWN Q3 /Q2 Q1 Q0

SETF /PHI90
CLOCKF CLK
CHECK S1 S2 /S3 S4 UP DOWN Q3 Q2 /Q1 /Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 /UP DOWN Q3 Q2 /Q1 /Q0

SETF /PHI0
CLOCKF CLK
```

```
CHECK /S1 S2 /S3 /S4 UP DOWN Q3 Q2 /Q1 Q0

CLOCKF CLK
CHECK /S1 /S2 /S3 /S4 /UP DOWN Q3 Q2 /Q1 Q0

SETF PHI90
CLOCKF CLK
CHECK /S1 /S2 S3 /S4 UP DOWN Q3 Q2 Q1 /Q0

CLOCKF CLK
CHECK /S1 /S2 S3 S4 /UP DOWN Q3 Q2 Q1 /Q0

SETF PHI0
CLOCKF CLK
CHECK S1 /S2 S3 S4 UP DOWN Q3 Q2 Q1 Q0

CLOCKF CLK
CHECK S1 S2 S3 S4 /UP DOWN Q3 Q2 Q1 Q0

SETF /PHI90
CLOCKF CLK
CHECK S1 S2 /S3 S4 UP DOWN /Q3 /Q2 /Q1 /Q0

CLOCKF CLK
CHECK S1 S2 /S3 /S4 /UP DOWN /Q3 /Q2 /Q1 /Q0

SETF /PHI0
CLOCKF CLK
CHECK /S1 S2 /S3 /S4 UP DOWN /Q3 /Q2 /Q1 Q0

SETF LD /D3 /D2 /D1 D0
CLOCKF CLK
CHECK /Q3 /Q2 /Q1 Q0

SETF /D3 /D2 D1 D0
CLOCKF CLK
CHECK /Q3 /Q2 Q1 Q0

SETF /D3 D2 /D1 D0
CLOCKF CLK
CHECK /Q3 Q2 /Q1 Q0

SETF /D3 D2 D1 D0
CLOCKF CLK
CHECK /Q3 Q2 Q1 Q0

SETF D3 /D2 /D1 D0
CLOCKF CLK
CHECK Q3 /Q2 /Q1 Q0

SETF D3 /D2 D1 D0
CLOCKF CLK
CHECK Q3 /Q2 Q1 Q0
```

```
SETF D3 D2 /D1 D0
CLOCKF CLK
CHECK Q3 Q2 /Q1 Q0

SETF D3 D2 D1 D0
CLOCKF CLK
CHECK Q3 Q2 Q1 Q0

;DESCRIPTION
;
;THIS PAL20X10 IMPLEMENTS A TWO CHANNEL SHAFT ENCODER WITH AN
;INTERNAL 4-BIT UP/DOWN COUNTER.
;
;BOTH THE "UP" AND "DOWN" OUTPUTS OF THE PAL DEVICE ARE NORMALLY
;HIGH.
;
;WHEN THE SIGNAL AT THE "PHI0" INPUT LEADS THE SIGNAL AT THE
;"PHI90" INPUT, THE "DOWN" OUTPUT ALTERNATES BETWEEN HIGH AND
;LOW LEVELS AND THE COUNTER WILL COUNT DOWN.  WHEN THE SIGNAL
;AT THE "PHI0" INPUT LEADS THE SIGNAL AT THE "PHI90" INPUT,
;THE "UP" OUTPUT ALTERNATES BETWEEN HIGH AND LOW LEVELS AND
;THE COUNTER WILL COUNT UP.
;
;INPUT "X4" SELECTS BETWEEN HALF (X4=H) OR QUARTER (X4=L) CLK
;FREQUENCY OF THE COUNTER OUTPUTS.
;
;THE INTERNAL 4-BIT SYNCHRONOUS COUNTER HAS COUNT UP, COUNT DOWN
;CAPABILITIES.  ALSO, THE COUNTER CAN PARALLEL LOAD AND HOLD DATA
;INDEPENDENTLY OF THE SHAFT ENCODER SECTION.  THE REGISTERS ARE
;SYNCHRONOUSLY INITIALIZED WHEN /SSET IS HELD LOW.
;
;THE CONTROL INPUTS PROVIDE THESE OPERATIONS WHICH OCCUR
;SYNCHRONOUSLY AT THE RISING EDGE OF THE CLOCK.
```

IMPORTANT
NOTES

Military Applications

Monolithic Memories/Advanced Micro Devices is a major participant in the Command, Communication and Control (C^3) military marketplace. Applications include programmable obsolescence solutions, surface mount designs, radar systems, missile guidance, aircraft graphic processors, microprocessor support logic, state-machine designs, military computer hardware, redundant solutions for backup systems, and reconfigurable hardware using Logic Cell Arrays.

Programmable products offer an excellent solution to military obsolescence difficulties. The dynamics of the semiconductor industry has shortened product life cycles, creating a sunset technology market. Programmable products offer a quick, low-cost replacement for obsolete parts, through direct pin compatibility or board emulation.

MMI/AMD has long been a leader in military bipolar PAL, PROM and FIFO products. We will soon be introducing military zero-power CMOS PAL devices and military CMOS Logic Cell Arrays. With pin and function compatible PAL products offered in both bipolar and CMOS technologies, Monolithic Memories offers backup systems (redundant solutions) in case of a failure with the original design. So, whether the customer needs high-speed bipolar or low-power CMOS, we will be the only company offering both technologies for the military marketplace.

Surface mount technology is an assembly technique when circuit boards are assembled with components bonded to metal pads on the board surface, instead of being inserted into through-holes like conventional Dual In-line packages (DIP). Surface mount components can be mounted on both sides of a circuit board, doubling the functional density. This lowers cost by reducing the number or size of boards required. The military need for smaller or denser-packed systems make surface mounting a must. We meet this challenge offering a full line of products in surface mount packaging, including Cerpack (W) and Leadless Chip Carriers (LCC).

For more information on military products and applications, see the military datasheet section (page 5-415).

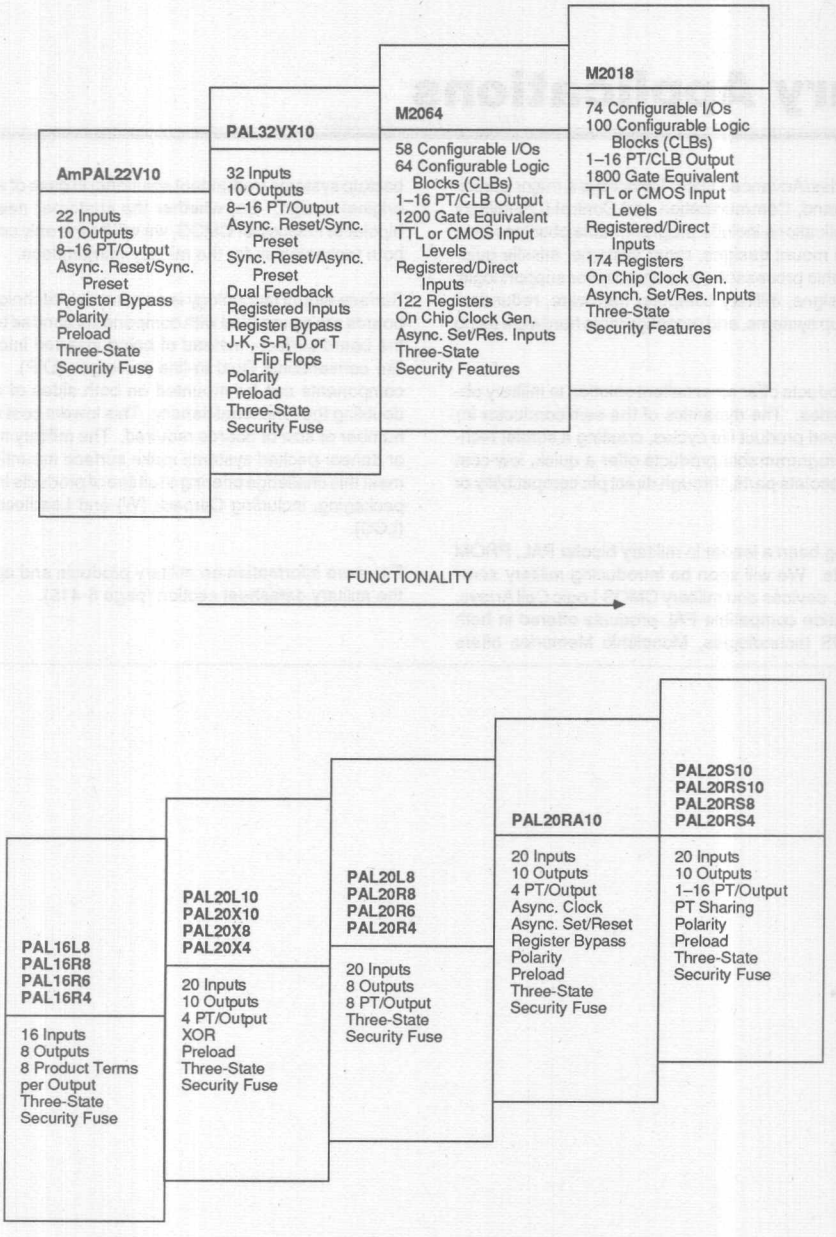


Figure 1. Military Configurable Architectures

Radiation Hardness

Spurred on by America's Strategic Defense Initiative, military equipment manufacturers are looking for radiation hardened parts to put into strategic weapon and space systems. It has been stated that some level of radiation tolerance will be required in up to 50% of all military applications by 1990. Due to this increased need for radiation-resistant integrated circuits, MMI and AMD have embarked on a program to determine what radiation dose rates its products will withstand, and to provide increasingly radiation-tolerant products.

The Galileo probe to Jupiter was bombarded with 100,000 rads (Si) within a one hour period. In contrast, the earth's surface absorbs 0.1 rads in a year. Military aircraft must withstand a minimum of 2,500 rads total dose. A nuclear explosion emits transient- ionizing radiation of at least 10^9 rads (Si) per second. Radiation dosage and type varies greatly with the environment. Table 1 shows radiation dosages and semiconductor tolerances.

Radiation effects can be divided into two major categories; natural space effects and man-made nuclear effects.

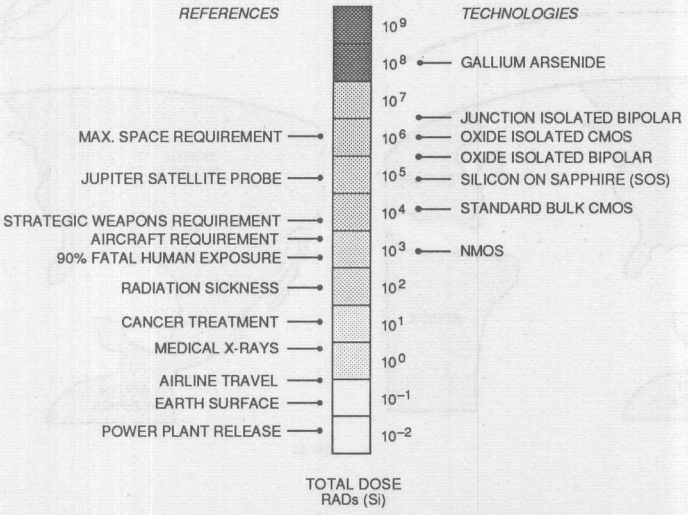
Space radiation consists mainly of cosmic rays and charged particles in radiation belts (Figure 1). Cosmic rays from the sun, made up of protons and nuclei, cause single event upsets (SEU). A SEU occurs when a high-energy particle passes through an integrated circuit, changing an internal logic state.

Radiation belts around planets, such as the Van Allen Belt circling the earth, carry trapped protons and electrons. These charged particles accumulate over time in CMOS devices, causing threshold voltage shifts. In oxide-isolated bipolar circuits this ionizing radiation (total dose) will cause an increase in leakage current.

A nuclear explosion irradiates a large neutron fluence and high energy gamma rays (Figure 2). Neutron radiation damages the semiconductor lattice structure, creating recombination sites for minority carriers, which decreases bipolar transistor current gain. Neutrons induce no change to CMOS devices, which are majority-carrier structures.

Nanoseconds after a nuclear blast, short, high-intensity electromagnetic pulses lash out. Difficult to stop, these gamma rays generate huge numbers of holes and electrons, causing large currents, destroying both bipolar and CMOS circuits. CMOS is especially susceptible due to its inherently low latch-up sensitivity of 10^8 rads per second.

Emerging technologies and process improvements will make the basic military hardness requirements for semiconductor devices a reality (Table 2). Gallium Arsenide (GaAs) products now meet a no-upset dose rate of 5×10^{11} rads per second and a total dose of 10^8 rads. While junction-isolated bipolar devices also meet inherently high dose rates of up to 10^{10} rads/second and total doses



438 01

Table 1. Radiation Perspectives

Natural

Cosmic Rays

- Protons and Nuclei
- Causes Single Event Upsets (SEUs)

Charged Particles

- Electrons and Protons Found in Radiation Belts
- Accumulated Ionizing Radiation Over Time (Total Dose) Measured in rads (Si)
- Causes Accumulation of Trapped Charges → Shifting Threshold Voltages in CMOS
- Increases Leakage Currents in Bipolar

Nuclear Effects Man-Made

Neutrons

- Neutron Fluence Measured in Neutrons Per Square Centimeter
- Damages Lattice Structure → Decreases Gain
- Affects Bipolar More Than CMOS

Gamma Rays

- Transient-Ionizing Radiation (Dose Rate) Measured in rads (Si) Per Second
- High Energy Photons of 10^9 rads Per Second
- Causes Large Photocurrents → Increases Supply Current
- Results in Latch-Up and Permanent Damage from Burn-Out
- Difficult to Shield Against
- Ruins CMOS

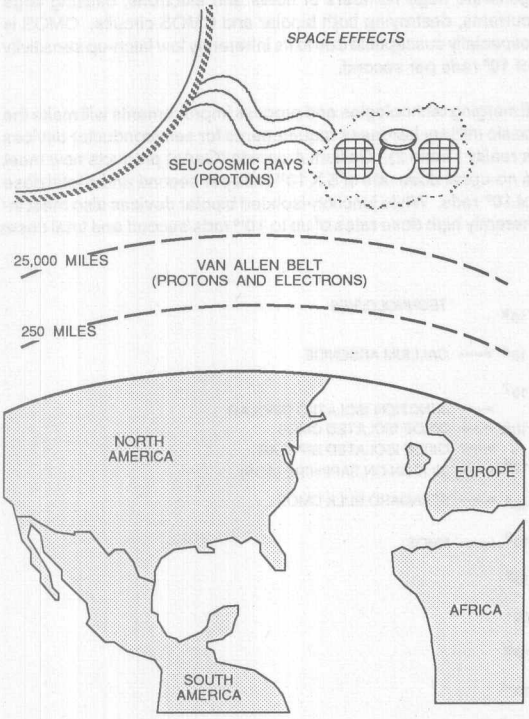


Figure 1. Space Effects

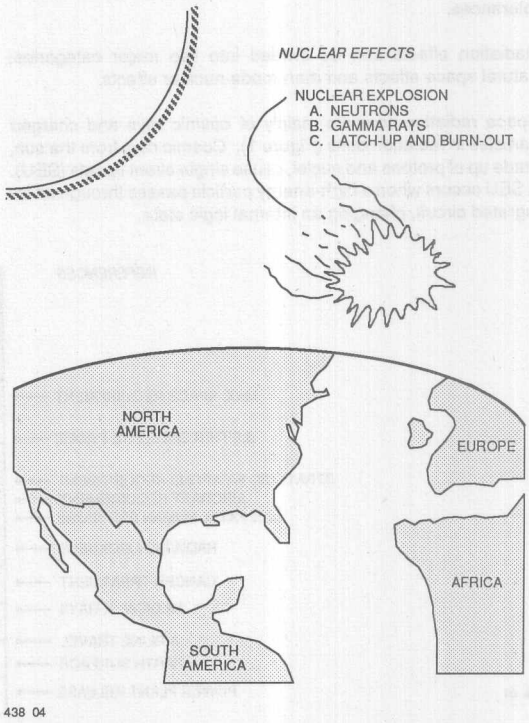


Figure 2. Nuclear Effects

of up to 10^7 rads, radiation-hardened CMOS circuits are not far behind with a typical dose rate of 10^{10} rads/second and a total dose of 10^6 rads (Table 3).

Process changes to improve the radiation tolerance of CMOS devices are not without penalties. Guard rings reduce latch-up sensitivity and leakage paths, but decrease circuit density. Thinner gate oxides increase radiation resistance but cause reliability problems due to hot electron effects. A dry oxide process produces less defects but increases manufacturing time. Done correctly, these sacrifices will produce a marketable radiation-hardened CMOS process.

RADIATION TYPE	AIRCRAFT	NUCLEAR	SPACE
Dose Rate with no upset (rads/second)	10^8	10^{11}	—
Dose Rate with no latch-up (rads/second)	10^{10}	10^{12}	—
Total Dose (rads (Si))	2500	5000	10^6
Neutron Fluence (neutrons/square centimeter)	100	10^{13}	—
Single Event Upset involving electrons (bits/square centimeter)	—	10^{-10}	—
Single Event Upset involving protons (bits/square centimeter)	—	—	10^{-14}

Table 2. Basic Military Hardness Requirements

At Monolithic Memories, radiation data has been obtained for neutron fluence and dose rate effects. Products representing bipolar junction and oxide-isolated processes passed neutron irradiation levels of up to 10^{13} neutrons per square centimeter (Table 4). Dose rate data obtained on the junction-isolated products showed recovery in fifty to seventy microseconds from a one microsecond pulse of 2 X 10^{10} rads (Si) per second (Table 5). MMI and AMD will move from radiation testing towards new radiation-tolerant designs and processes as the need for radiation-hardened products increases.

Space Effects

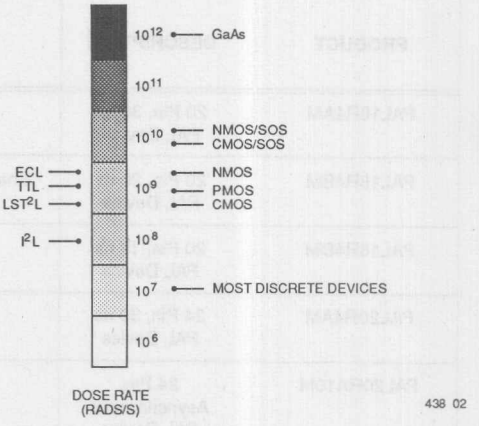


Table 3. Typical Ionizing Dose Rate Upset Levels

PRODUCT	DESCRIPTION	PROCESS	PASSED DATASHEET LIMITS UP TO (neutrons/cm²)	PASSED FUNCTIONAL UP TO (neutrons/cm²)
PAL16R4AM	20 Pin, 30 ns PAL Device	Junction Isolated Diffused Bipolar	4.5×10^{13}	1×10^{14}
PAL16R4BM	20 Pin, 20 ns PAL Device	Shallow-Junction Isolated Implanted Bipolar	4.5×10^{13}	1×10^{14}
PAL16R4DM	20 Pin, 15 ns PAL Device	Oxide Isolated Bipolar	4.5×10^{13}	1×10^{14}
PAL20R4AM	24 Pin, 30 ns PAL Device	Junction Isolated Diffused Bipolar	8×10^{13}	1×10^{14}
PAL20RA10M	24 Pin, Asynchronous PAL Device	Junction Isolated Diffused Bipolar	9.6×10^{13}	1×10^{14}
PAL32R16M	1500 Gate, 40 Pin MegaPAL™ Device	Junction Isolated Diffused Bipolar	1×10^{14}	1×10^{14}
53S1681	16K PROM	Junction Isolated Diffused Bipolar	4.5×10^{13}	4×10^{13}
53RA1681A	16K Registered PROM	Junction Isolated Diffused Bipolar	8×10^{13}	1×10^{14}
53S3281	32K PROM	Junction Isolated Diffused Bipolar	1×10^{14}	1×10^{14}

Table 4. Neutron Bombardment Data

PRODUCT	DESCRIPTION	PROCESS	DOSE RATE (rads/second)	PHOTOCURRENT (amps)
PAL12H6M	20 Pin Small PAL Device	Junction Isolated Diffused Bipolar	2.6×10^{10}	4.2
PAL14H4M	20 Pin Small PAL Device	Junction Isolated Diffused Bipolar	2.1×10^{10}	3.9
PAL10L8M	20 Pin Small PAL Device	Junction Isolated Diffused Bipolar	2.5×10^{10}	4.2
PAL12L6M	20 Pin Small PAL Device	Junction Isolated Diffused Bipolar	2.2×10^{10}	3.8
PAL14L8M	24 Pin Small PAL Device	Junction Isolated Diffused Bipolar	1.2×10^{10}	3.6
PAL16L6M	20 Pin Small PAL Device	Junction Isolated Diffused Bipolar	2.3×10^{10}	4.0
PAL16L8M	20 Pin Standard PAL Device	Junction Isolated Diffused Bipolar	2.5×10^{10}	5.3
PAL16R8M	20 Pin Standard PAL Device	Junction Isolated Diffused Bipolar	1.9×10^{10}	5.3
PAL16R6M	20 Pin Standard PAL Device	Junction Isolated Diffused Bipolar	2.4×10^{10}	5.7
PAL16R4M	20 Pin Standard PAL Device	Junction Isolated Diffused Bipolar	2.1×10^{10}	5.6
PAL20L10M	24 Pin Exclusive-OR PAL Device	Junction Isolated Diffused Bipolar	2.1×10^{10}	6.7
PAL20X10M	24 Pin Exclusive-OR PAL Device	Junction Isolated Diffused Bipolar	1.9×10^{10}	5.9
PAL20X8M	24 Pin Exclusive-OR PAL Device	Junction Isolated Diffused Bipolar	2.1×10^{10}	5.8

Table 5. Transient Upset Test Data

IMPORTANT
NOTES

Article Reprints

2

Article Reprints

Programmable Logic Device Preserves Pins, Product Terms (PAL32VX10) 2-589
PLD Programmability Extends its Sway Over Complex I/O (PALC29M/MA16) 2-593
PROSE Devices Simplify State Machine Design (PMS14R21) 2-601
Designing a State Machine With a Programmable Sequencer (PMS14R21) 2-607
FPCs and PLDs Simplify VME Bus Control (Am29PL141) 2-619
Fuse-Programmable Chip Takes Command of Distributed Systems (Am29PL141) 2-633
PAL Device Buries Registers, Brings State Machines to Life (AmPAL23S8) 2-639
Programmable Event Generator Conquers Timing Restraints (Am2971) 2-644
Wait-State Remover Improves System Performance (PAL22V10) 2-648
PLDs Implement Encoder/Decoder for Disk Drives (PAL22V10) 2-651
Mixing Data Paths Expands Options in System Design (PAL22V10) 2-660
Programmable Logic Chip Rivals Gate Arrays in Flexibility (PAL22V10) 2-670
XOR PLDs Simplify Design of Counters and Other Devices (PAL20X10) 2-676
The PAL20RA10 Story —The Customization of a Standard Product (PAL20RA10) 2-683
PLDs Abound: RAM-Based Logic Joins In 2-699
Introduction to Programmable Array Logic 2-702
Logical Alternatives in Supermini Design 2-711

Conference Proceedings

New PAL Device Architecture Extends Design Flexibility (PAL32VX10) 2-719
PROSE Architecture and Design Methodology (PMS14R21) 2-724
Blazing Fast PAL Devices Enable New Application Areas (PAL16R8-10, PAL10H20P/G8) 2-730

IMPORTANT
NOTES

Programmable Logic Device Preserves Pins, Product Terms

Chris Jay
Monolithic Memories Inc., 2175 Mission College Blvd., Santa Clara, CA 95054; (408) 970-9700.

Buried registers and exclusive-OR gates make this PAL device a sound foundation on which to build state machines that conserve I/O pins.

Programmable-logic devices continue to evolve, responding to designers' demands for improved performance, functionality, and flexibility. Among the newest is the PAL32VX10, a programmable-array logic (PAL) device that enhances the architecture of the earlier PAL22V10. The result is a more flexible general-purpose device.

As a functional and architectural superset of the 22V10, the 32VX10 can be programmed to carry out any function that its predecessor can. It also adds two features: registers that can be buried for designing state machines, and exclusive-OR (XOR) gates for counter and JK functions (ELECTRONIC DESIGN, Jan. 8, p. 45).

The number of inputs to the fuse array of the latest programmable-array logic (PAL) device has been increased to 32 by the addition of ten buried feedback paths. A macrocell output programmed as an input accesses the chip's feedback path, which is buried within the fuse array.

As a result, buried state machines can be configured in the device. Any logic design calling for many input pins can be configured with dedicated input pins and any of the of macrocell I/O pins (Fig. 1). This option leaves pins free for configuring outputs as inputs, and other Boolean functions may be realized in buried registers. Thus, functionality of the macrocell need not be lost.

The macrocell's XOR gate (denoted by the X in the part number) simplifies state-machine design because it promotes gate product-term economy. This economy is the ability to create complex state-machine designs without exhausting the limited number of product terms in each macrocell.

For example, there are ten D-type registers in the 32VX10, one in each of its ten macrocells. The XOR input to each register and no more than three product terms transform a D-type register to a JK register. Once the JK register is configured, subset functions like the T (toggle) and an SR (set-reset) may be derived.

The macrocell's product-term economy can be a result of the hold state inherently defined in the JK truth table—a function not offered by the D-type register alone (see the table). For a D-type register to hold its contents in a "set" state during several states, additional product terms must hold that condition until it is relaxed. In many tasks, applying product terms as holding functions could exhaust the limited supply in a given cell and compromise a state-machine design.

A practical task for the hold function is in a line-sync generator. The blanking and line-sync pulses are programmed as JK functions, and a hold condition is invoked for their active duration.

The XOR gate also influences the output polarity of a logic signal. Active-high or -low logic signals may be created at the output pins. In PAL devices without polarity control, the output can be inverted by applying DeMorgan's Theorem to a sum-of-products architecture. This approach, however, brings with it the added baggage of more product terms. For example, one term containing N literals (Boolean variables) is inverted by creating the sum of N negated literals, with each negated literal using a product term.

Each of the 32VX10's ten macrocells is configured around a sum-of-products and XOR input to a D-type register, which is clocked from a dedicated input pin. Each macrocell can be programmed for a combinational or registered output. One product term, associated with the control to bypass the register, selects the path dynamically, if needed.

Typically, the registers are programmed as a refresh counter for dynamic RAM. When a host microprocessor must access the RAM for read and write cycles, its address-line outputs bypass the registers, which contain the current refresh count for the RAMs. Instead, the outputs address the memory inputs directly. Afterwards, the address outputs

Reprinted with permission from Electronic Design,
Vol. 35, No. 13 (May 28, 1987). Copyright 1987 Hayden Publishing Co., Inc.

from the registers are enabled again to refresh the RAM.

In fast video applications, a similar philosophy is in effect. The host microprocessor bypasses the contents of the video-memory address counter to update the video RAM. It is important, though, that any counter-increment signal be disabled when the registers are bypassed; an active clock input would overwrite any contents of the D-type registers. In a video RAM system, this might not be a disadvantage, but for dynamic-RAM arrays, overwriting a refresh counter could lead to loss of data.

One product term might also dynamically influence the logic signal's output polarity. With a registered output, a dedicated register-output-polarity multiplexer selects either a Q or $\overline{Q}$ output, routing the selected signal to the output pin. One programmable polarity fuse selects the required path. This fuse does not affect the programmable polarity associated with combinational outputs.

The global product terms from the fuse array are dedicated to resetting the ten registers synchronously and also presetting them asynchronously. The action of these two product terms can be selected by the programmable-polarity multiplexer for individual registered cells.

For instance, for a registered output, the asynchronous-preset gate can be programmed to a synchronous-reset gate or vice versa. The single product-term input to the XOR gate can select the output polarity for either combinational or registered outputs.

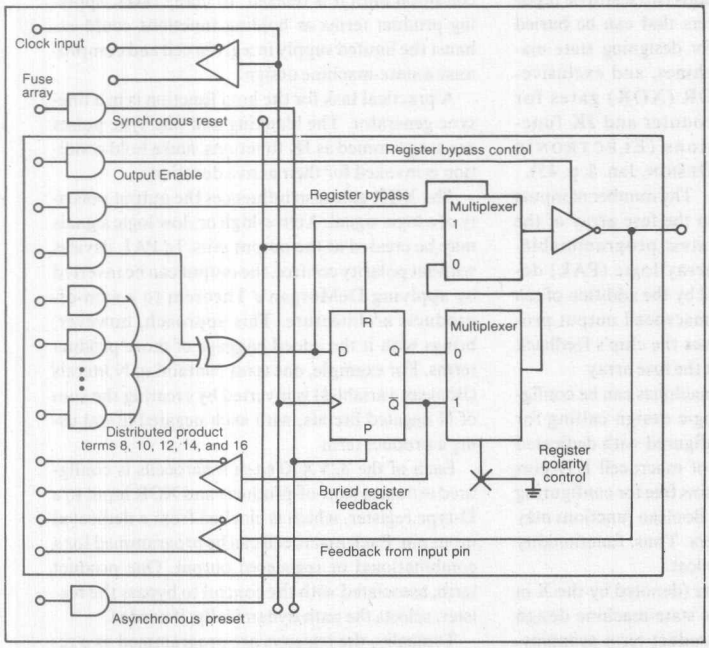
The second added feature of the new PAL device, its conversion of an internal D-type register to a JK-register cell, is demonstrated by five Boolean equations formed from a sum of products:

$$\underline{Q} := \overline{\underline{Q}} * J + Q * \overline{K} \quad (1)$$
$$\underline{Q} := \overline{\underline{Q}} * \overline{J} + Q * K \quad (2)$$
$$\underline{Q} := \underline{Q} :+: \overline{\underline{Q}} * J + Q * K \quad (3)$$
$$\underline{Q} := \underline{Q} :+: \overline{\underline{Q}} * J + Q * \overline{K} \quad (4)$$
$$Q := \overline{Q} :+: \overline{Q} * \overline{J} + Q * \overline{K} \quad (5)$$

where the Boolean operators are:

— for an inversion;
:+: for an XOR operation;
+ for an OR operation;
* for an AND operation; and
:= is the state-machine operator "becomes equal to"

Note that all five Boolean equations have true or complement conditions for either J or K; and a mixture of J and $\overline{K}$ or $\overline{J}$ and K functions. Three of the equations require the XOR function. Moreover, J and K active-high conditions can be used in equation 3 only with an XOR operator.



1. Each of the ten macrocells in Monolithic Memories' PAL32VX10 PAL device is driven by an exclusive-OR gate, a feature that promotes frugal use of product terms. For example, the gates reduce the number of terms needed to implement JK registers. The macrocells can also feed output and input signals back to buried registers. As a result, designing state machines with the device is easier.

The Boolean operators are compatible with the Palasm assembler-support software. Palasm is a computer-aided (CAD) tool that generates programming information for PAL devices. The Boolean expression information describes the operation of the device to be programmed.

A good design specification done by a CAD tool like Palasm should come with a complete comment field to describe the function of the Boolean equations. Palasm accepts a comment field in the description of a logic-design file. When a semicolon precedes a statement, Palasm ignores that statement during the assembly process.

The functional description of the logic circuits is in terms of Boolean statements. A later version of Palasm will accept

state-machine logic descriptions on a higher level. Then, designers will be able to deal with logic statements containing JK functions without describing them as a series of a sum of products, as is now the case.

The ability to create subsets of a JK register gives the logic designer more flexibility to design sequential-logic circuits. One of the subset functions, for example, is the toggle (T) flip-flop, which lends itself to counter applications. Binary counters require that the contents of any general register, Q_N, remain in a hold state until the product term of all the lesser significant registers goes to a logic high. Then the register must change state or toggle after the arrival of the next clock pulse. The T function may be created by tying the J and K inputs together and treating this composite as the T input. A hold state is maintained when T = 0, and a change of state results when T = 1.

The application of the toggle function to binary counters can be developed by considering a general register Q_N in a binary count sequence, where the lesser significant registers are $Q_{N-1}, Q_{N-2}, \ldots Q_1, Q_0$. Using equation 3, the JK description for a general register is given as

$$Q_N := Q_N :+: K * Q_N + J * \overline{Q}_N \quad (6)$$

A hold condition must apply when the product term of all lesser significant registers is a logic zero, or:

$$Q_{N-1} * Q_{N-2} * \ldots Q_1 * Q_0 = 0$$

and a toggle condition must apply when

$$Q_{N-1} * Q_{N-2} * \ldots Q_1 * Q_0 = 1$$

This develops to

$$J = K = Q_{N-1} * Q_{N-2} * \ldots Q_1 * Q_0 \, (7)$$

which represents the T input to a toggle register. Substituting equation 7 into 6 leads to

$$Q_N := Q_N :+: Q_N * (Q_{N-1} * Q_{N-2} \ldots Q_1 * Q_0) + \overline{Q}_N * (Q_{N-1} * Q_{N-2} * \ldots Q_1 * Q_0) \, (8)$$

Substituting equation 7 into 6 for J and K inputs leads to

$$Q_N := Q_N :+: (Q_N + \overline{Q}_N) * Q_{N-1} * Q_{N-2} * \ldots Q_1 * Q_0 \, (9)$$

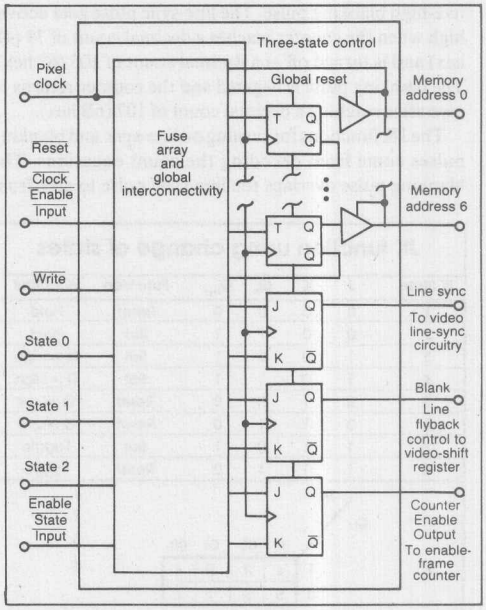
Because $(Q_N + \overline{Q}_{\overline{N}}) = 1$, equation 9 reduces to

$$Q_N := Q_N \, (\text{hold } Q_N) :+: Q_{N-1} * Q_{N-2} * \ldots Q_1 * Q_0 \, (\text{toggle } Q_N) \, (10)$$

Equation 10 is a general equation for a register in a binary counter. Only two product terms are used because the XOR gate makes it possible to invoke the toggle function as a JK subset. Without the XOR gate, the general equation for register N would be

$$Q_N := \overline{Q}_{\overline{N}} * Q_{N-1} * Q_{N-2} * \ldots Q_1 * Q_0 \, (\text{toggle } Q_N) + Q_N * \overline{Q}_{\overline{N-1}} \, (\text{hold } Q_N) + Q_N * \overline{Q}_{\overline{N-2}} \, (\text{hold } Q_N) \; . \; . \; . + Q_N * Q_1 \, (\text{hold } Q_N) + Q_N * Q_0 \, (\text{hold } Q_N) \, (11)$$

Equation 11 requires N product terms as hold functions. This means that for a 10-bit counter, the most significant register, Q_9, would need nine individual product terms as hold functions and one additional product term for a toggle function.



2. Thanks to its easy implementation of the JK function and its subsets, the 32VX10 makes a compact line-address counter and line-sync generator for video tasks.

In contrast, equation 10 makes use of the XOR function, requiring one product term to hold and one product term to toggle. No matter how significant the register in the count sequence, the inclusion of an XOR function in the equation eliminates large numbers of product terms. The advantage of the XOR gate increases for a 10-bit, up-down counter because the required number of product terms doubles. Q_0 would normally need 20 product terms but with an XOR gate, only three do the job.

A practical outcome of configurable JK and T registers is the design of a line-sync generator circuit for controlling a video RAM, and for synchronizing line-display information to a CRT interface (Fig. 2). Memory-address outputs MA_0 through MA_6, which are derived from an internal counter, drive the address inputs to the video RAM. One complete line of video information is displayed and surrounded by line-sync and blanking pulses from two macrocells programmed as JK registers.

The internal memory-address counter is programmed to address an 80-character screen and then issue an active-high blanking pulse. The line-sync pulse goes active-high when the counter reaches a decimal count of 79 (4F hex) and is turned off at a decimal count of 103 (67 hex). The blanking pulse is negated and the counter returns to zero after reaching a decimal count of 107 (6B hex).

The JK functions for turning on line-sync and blanking pulses come from decoding the count equations. The blanking pulse overlaps the line-sync pulse to give front and back porches to the video line-scan signal. Also, because the host shares access to the video memory, it can update video information. In that case, a Write ($\overline{WR}$) is driven low to produce three-state video-address outputs.

When at work, the microprocessor can change the contents of the video RAM during frame- or line-signal flyback periods to avoid displaying random data during active display periods. To make such changes possible, the $\overline{WR}$ input can be derived from a smaller combinational PAL device that decodes the microprocessor's address lines and maps the video RAM into its address space.

An asynchronous reset input ($\overline{RST}$) powers up the PAL to a known state. With its Count Enable Input ($\overline{CEI}$) and Output ($\overline{CEO}$) signals, the device can cascade with other circuitry, such as a frame-sync generator and a video-shift register.

The memory-address counters are programmed as T flip-flops; line-sync, blanking and $\overline{CEO}$ functions as JK registers. An Enable-State Input ($\overline{ENST}$) enables input states 0 through 2 as count enable inputs in preference to the $\overline{CEI}$ signal. When in the low state, $\overline{ENST}$ qualifies input states 0 through 2 as active-high count-enable inputs.

To complement the line-sync generator design, the 32VX10 is programmed as a 7-bit video-shift generator. Seven registers shift video data and three are programmed as a 3-bit counter to control loading and shifting of video data in the shift register. When all are high, counter output states 0 through 2 cause the shift register to load and the line-sync generator to increment.

For the example circuit, another 32VX10 is programmed to support the line-sync generator. As with the other 32VX10s, the frame-sync output is programmed as a JK function. The internal counter accesses row-character information as well as the contents of the video RAM. The 32VX10 video-shift register accepts dot information from the character generator and supplies a parallel-to-serial output for the video display. Any parallel-to-serial output format can be handled by the 32VX10.

Readers interested in the PAL design specifications may contact the author for further details.□

Chris Jay, logic-cell-array applications manager with Monolithic Memories, has been with the company since 1983. Three years were spent as a field-application engineer in Northern Europe. Before that, he worked for Texas Instruments in Bedford, England and Fairchild Camera and Instrument in Bristol, England, performing logic design and applications engineering, respectively. Jay earned a BSc (honors) degree from Essex University in Coulchester, England in 1977.

JK function using change of states

JK State	J	K	Q_N	Q_{N+1}	Function	JK Output
1	0	0	0	0	Reset	Hold
2	0	0	1	1	Set	Hold
3	1	0	0	1	Set	Q := High
4	1	0	1	1	Set	Q := High
5	0	1	0	0	Reset	Q :=Low
6	0	1	1	0	Reset	Q :=Low
7	1	1	0	1	Set	Toggle
8	1	1	1	0	Reset	Toggle

Q \ JK	11	01	01	00
1	s	R	R	s
0	S	S	r	r

1.
2.
3.
4.

Q = Q :+: /Q*J+Q*K
/Q = /Q :+: /Q*J + Q*K
Q = /Q :+: /Q*/J +Q*/K

PLD Programmability Extends Its Sway Over Complex I/O

Electrically erasable CMOS parts, the densest 24-pin SLIM PALs yet, reach new heights in architectural flexibility

Om Agrawal, Product Planning Manager
Fares Mubarak, Senior Test Engineer
Field-Programmable Logic Division
Advanced Micro Devices, Inc., Sunnyvale, CA

Two electrically erasable PAL devices being introduced here by Advanced Micro Devices offer by far the greatest flexibility of any programmable array logic (PAL) device to date. Several features contribute to the adaptability of these CMOS ICs—the highest yet number of product terms (188 and 178) of any 24-pin PAL; the optimization of one device, the AmPAL29M16, for synchronous designs and the other, the AmPAL29MA16, for asynchronous and glue logic applications; and the convenience of some 11,000 electrically programmable and erasable interconnecting cells (see *Figs. 1 and 2*).

But the most distinctive enhancement is the programmable input and output bidirectionality of each device's 16 *I/O logic macrocells* (I/OLMs). Until now, some *programmable logic devices* (PLDs) have had output macrocells, but none has had input macrocells, let alone the I/O type. The I/O pin on each I/OLM can in addition be configured as combinatorial, as an edge-triggered register, or as a transparent latch.

What's more, on 8 of the 16 I/O pins, the associated 8 register/latches can serve independently of the pins as programmable buried state registers. Current 40-pin CMOS EPLDs have only half that number. These internal registers offer designers an extra eight flip-flops for such jobs as keeping track of flags or generating timing and control signals.

The 11,000 or so interconnecting elements are *electrically erasable* (E^2) cells that define what each PAL device can do. They connect all array inputs to the programmable AND-OR array, and they configure the architecture of the I/OLMs. The

Reprinted from ELECTRONIC PRODUCTS Magazine—Vol. 29, No. 24, May 15, 1987
645 Stewart Ave., Garden City, NY 11530 • © 1987 Hearst Business Communications, Inc.

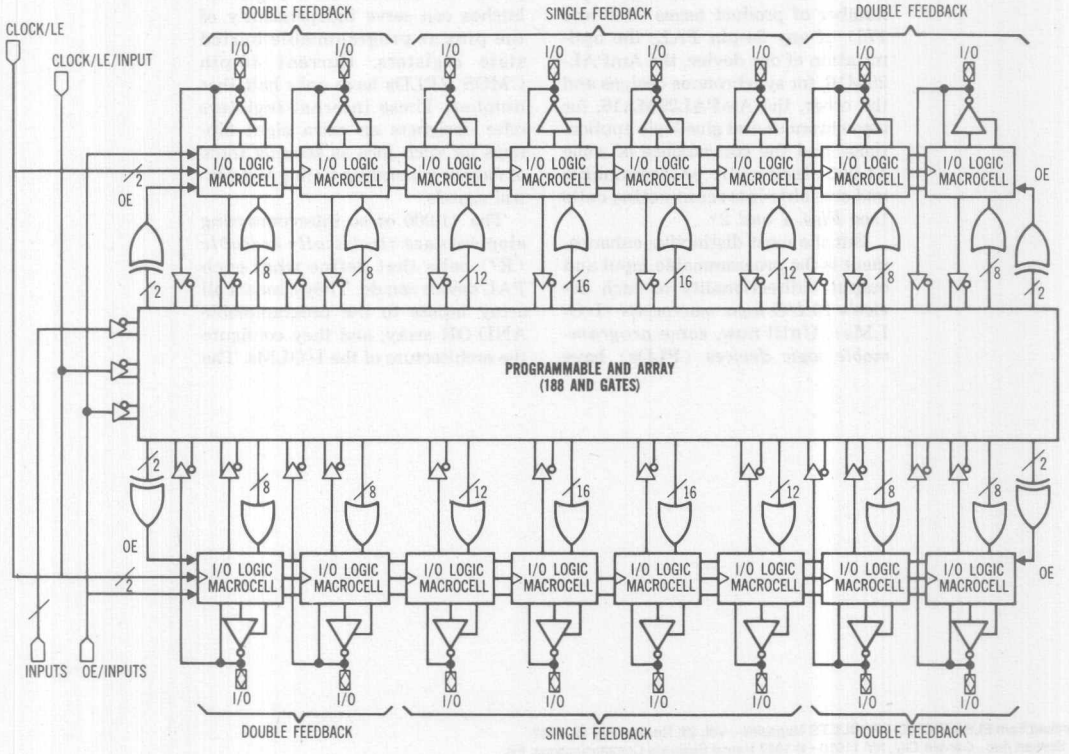
user programs these cells by charging and discharging them, so that design changes can be made quickly—far faster than for ultraviolet-light erasable PLDs, or EPLDs.

The record number of product terms, 16 macrocells, and horde of E^2 cells make the two newcomers the densest 24-pin SLIM PAL devices available, whether bipolar or CMOS. The ICs are five to six times denser than industry-standard, first-generation 20- and 24-pin EPLDs and E^2PLDs and twice as dense as second-generation 24-pin parts. In fact, these CMOS PALs are denser even than 40-pin bipolar PLDs.

Their 35-ns max input-to-output propagation delay and external clock frequency of 20 MHz max fixes the new CMOS devices in the medium speed range expected of bipolar units. Maximum active power dissipation is 120 mA. Each comes in two logic levels—CMOS HC and CMOS HCT logic for TTL systems. (The trademark PAL belongs to Monolithic Memories.)

From what follows, it will become clear that the basic I/O logic macrocell structure is the same for both the synchronous 29M16 and asynchronous 29MA16. Differences arise, though, in the I/OLM control mechanisms—their clocking, resetting and presetting, and output enabling. Also, the distribution of the product terms between logic and control functions is different for the two PALs. The testing and debugging features, though, are identical. In conclusion, an application of the 29M16 will be outlined—the use of nearly a dozen of these packages to build a 32-bit *direct memory access* (DMA) controller.

Fig. 1. The architecture of the synchronous AmPAL29M16 E^2 programmable logic device derives much of its flexibility from its 16 bidirectional input and output logic macrocells. The macrocells may use either one or two clocks, and each controls the output of on average 11 logic product terms.



I/OLM with a difference

The flexibility of the I/O and storage elements of these parts lends them the look more of a gate array than a PAL device (see Figs. 1 and 2). Eight of the I/OLMs have single feedback to the AND/OR array, and eight have dual feedback. Single feedback comes from either the I/O pin or the storage element, dual feedback comes from both.

It's when an I/O pin is used for input that its associated storage element can be turned to use as a buried register or latch. As many as all eight of the dual-feedback I/OLMs can be devoted to user-programmable buried-state register/latches. These

are helpful for both synchronous and asynchronous state-machine designs.

The presence, number, and programmability of the E^2 cells within the I/OLMs is the most significant architectural variable of these PAL devices. (Second-generation PLDs had three to four times fewer architectural cells.) Is the macrocell to be an input or an output? An edge-triggered D flip-flop or a transparent latch? What are the macrocell's clocking, feedback, and output-enable schemes and output polarity? What if it frees up a buried state register?

Programming the E^2 cells

All these decisions are made by programming the E^2 cells. Single-feedback macrocells have nine E^2 cells, dual-feedback macrocells have eight. In its virgin, erased (charged or disconnected) state, an E^2 cell's value is 1. In its programmed (discharged or connected) state, it is a 0.

Both synchronous and asynchronous I/OLMs contain a configurable storage-element cell and five multiplexers—two 2:1 multiplexers for selecting I/O and feedback path and three 4:1 multiplexers for selecting output path, clock, and output enable (see *Fig. 3*).

Whether a macrocell is to be an input or output and the nature of its storage element (edge-triggered register or transparent latch) is set by E^2 cells S_2 and S_3. In its virgin state, the I/OLM is an output cell and the storage element is an edge-triggered register. It can be changed by the user to an input latch or register or output latch by appropriate programming of S_2 and S_3.

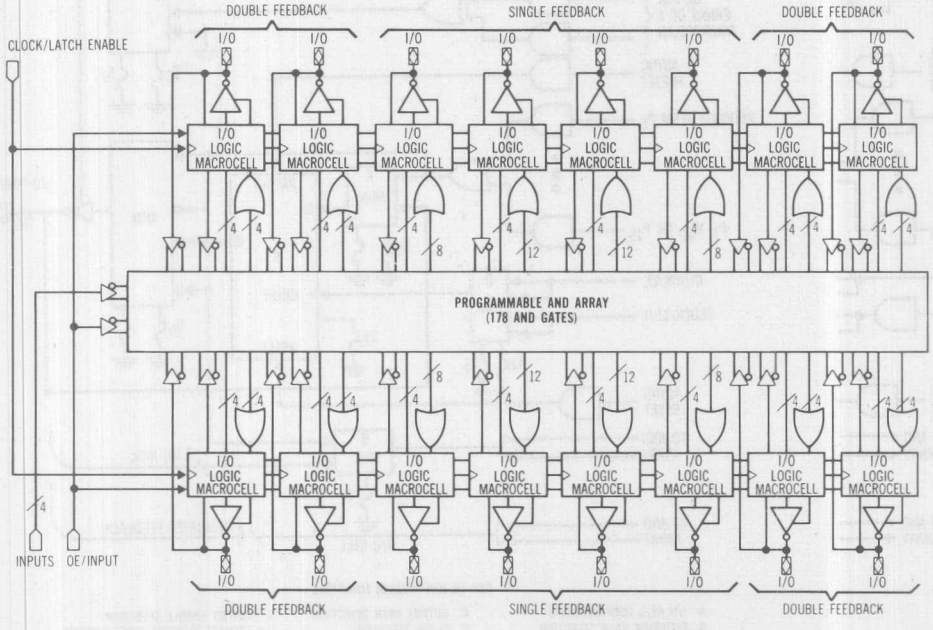
The polarity (active high or active low) of the cell's output and its sequential (registered or latched) or combinatorial nature are determined by S_0 and S_1. With both of these cells in the erased (charged) state, for example, the output is set to be active low and combinatorial.

By programming, or discharging, cell S_0, the designer makes the output go active high. The OR-gate output bypasses the storage element and feeds the output-path selection multiplexer directly.

On the other hand, by programming S_1, the designer makes the output sequential (registered/latched). The Q output of the storage element passes through the 4:1 multiplexer to an inverting output buffer and is fed back to the AND array internally. This configuration is particularly useful for state machine designs. Programming both S_0 and S_1 results in an active-high, sequential output.

The feedback multiplexer for the I/OLMs with single feedback is controlled by yet another E^2 cell (S_8). This separate control of the feedback multiplexer makes for the ultimate in flexibility in state machine design. Each I/OLM, with registered, combinatorial or latched outputs, choice of polarity, and dual feedback, has 12 possible output configurations.

Fig. 2. The 16 I/O macrocells in the asynchronous AmPAL29MA16 can each be allotted its own product-term-driven clock. The PAL's need for more control terms leaves an average of 7 logic product terms per macrocell.



The timing needs of synchronous state machines are much simpler than those of asynchronous and glue logic—hence the difference in the two PALs' clocking, output-enable, and preset/set arrangements and in the number of product terms assigned to logic and control.

For synchronous state-machine design, the flexibility of one or two synchronous clocks, preferably with polarity control, is often needed for individual macrocells. It is important for designing single-, dual-, or quadruple-state machines with a variable number of state registers. Programmable clock polarity offers the ability to trigger certain groups, or banks, of registers on one edge and other groups on the opposite edge. The result is faster internal state machines.

Early PAL devices had a single synchronous clock. Some newer synchronous CMOS erasable PAL devices offer dual clocks, but the two are selectable for a bank of 8 registers at a time, rather than for 16 macrocells individually. Thus, they are limited to, at most, two different state machines, each with an eight-register maximum.

In addition, these PAL devices usually have two separate clock pins. So, if the designer needs a single-state machine with up to 16 registers, the second clock pin is wasted.

Better clocking

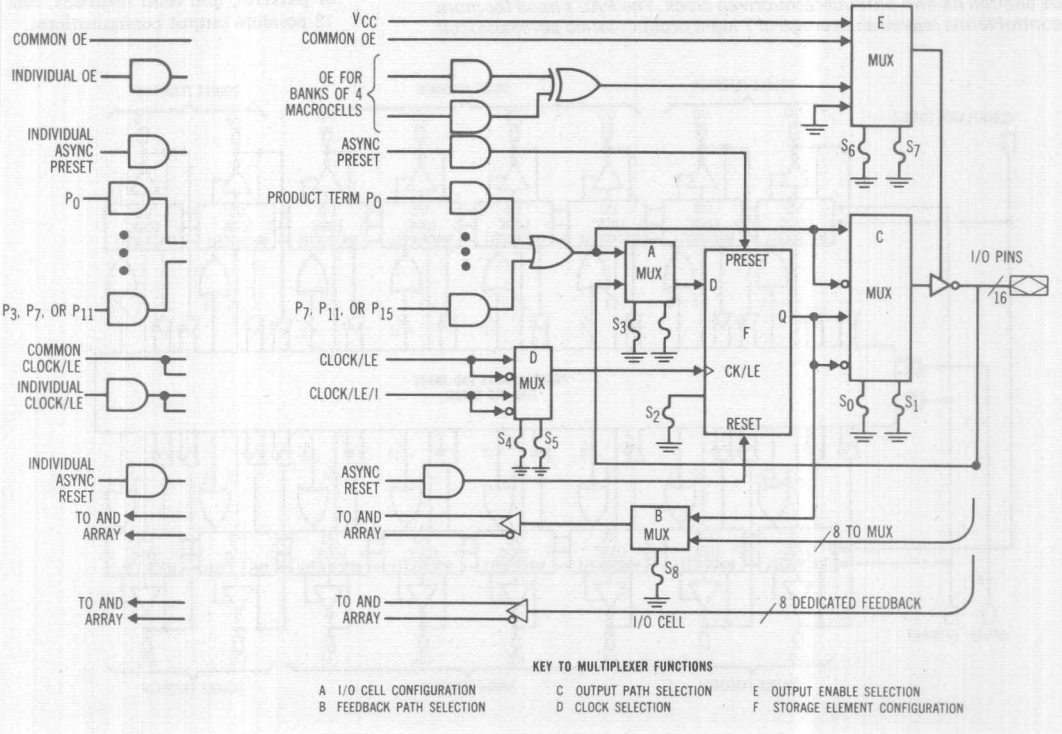
The 29M16s alleviate these drawbacks. Each I/OLM has a 4:1 programmable clock-selection mechanism selected by two E^2 cells, S_4 and S_5. For addressing synchronous logic, the 29M16 has one dedicated CLK/LE (latch enable) pin and one CLK/LE/I (input) pin. For macrocells configured as registers, these pins are clock pins; for latches, they're latch-enable pins.

Either the clock or latch-enable pin may be selected to trigger each storage element on either the rising or trailing edge. With this, system designers can implement up to a quadruple-state machine, with each meant to have a different number of registers.

Also, since the device offers 16 I/OLMs with input register/latches, the PAL is applicable to pipelined systems with eight input latches and eight output registers. The dual-clock capability allows the PAL to run with synchronous or asynchronous inputs. The clocks' programmable polarity allows the internal state machine to run at twice the speed of the external clock.

Further, since only one pin is dedicated as a clock/latch enable, a single state machine need not waste the

Fig. 3. The I/O logic macrocell, in either the synchronous AmPAL29M16 gate array (with input gates shaded yellow) or the asynchronous AmPAL29MA16 (inputs on yellow background at far left), depends for its functionality on how the erasable S cells configure the multiplexers and storage cell.



second clock pin. It can be an extra input.

While one or two synchronous clocks with programmable polarity are sufficient for synchronous state machines, asynchronous and glue logic applications have more demanding clock/latch requirements. They call for more flexible PAL devices with combinatorial outputs and feedback and with edge-triggered registers and transparent latches having individual clock/latch-enable capability (preferably with programmable polarity).

In addition, they require individual control of reset and preset and output-enable product terms, as well as slightly fewer product terms per output. Further, for asynchronous/glue logic applications, the clock-selection and output-enable schemes should be independent.

The 29MA16 addresses these needs with up to 16 product-term-driven asynchronous clock/latch-enable controls. Each of these controls is a product term driven from the AND-OR array as a function of certain input signals. These, in turn, are controlled by the user according to the application.

Like its synchronous counterpart, each I/OLM of the asynchronous 29MA16 has a 4:1 multiplexer for clock selection. Unlike the other, however, it allows the user to select from either a common synchronous clock/latch enable or an individual clock driven by a product term.

Preset and reset

The synchronous 29M16 uses common asynchronous reset and preset controls for all registered or latched inputs and outputs. When the product term for asynchronous preset is asserted, all the register/latches are immediately loaded with a high, independent of the clock. Conversely, when the asynchronous reset product term is asserted, all the register/latches are immediately loaded with a low, independent of the clock. Actual output depends on the macrocell polarity.

The asynchronous 29MA16, on the other hand, offers individual asynchronous reset and preset control for each of its storage elements. This feature, plus the individual clock capability, makes it easy for the device to replace a large amount of glue logic.

As in all PAL devices, the size, organization, and distribution of the internal product terms of the 29M16 and 29MA16's AND-OR array determine its logic capability. Both PAL devices can have 29 inputs, of which 24 issue from the I/O logic macrocells (8 from I/OLMs with single feedback and 16 from I/OLMs with dual feedback), 1 comes from the I/OE pin, and 3 are dedicated. The 29MA16 adds a fourth dedicated input, but the 29M16 instead has the CLK/LE/I pin.

Of the 29M16's 188 product terms, all of 176 are logic terms and 12 are control terms. The 176 logic product terms are distributed among the 16

PALs move to ever-greater flexibility

Important gauges of a PAL device's flexibility are the number and complexity of its input/output logic macrocells. With the introduction of its electrically erasable, CMOS-based AmPAL29M16 and AmPAL29MA16, Advanced Micro Devices has brought more flexibility than ever to programmable logic devices.

First-generation, bipolar fuse-programmable PAL devices had at most eight outputs fixed at the factory as either registered or combinatorial with active high or low polarity. The second generation improved on this somewhat by adding 10 programmable outputs configurable as registered or combinatorial and active high or low.

Although some second-generation CMOS EPROM-based PLDs offer as many as 16 logic cells, these can be configured only as outputs. Further, these outputs lack a transparent-latch capability, a requirement for both asynchronous and synchronous designs.

Architectural flexibility also extends to features such as variable product terms, varying the number of product terms per output, and programmable buried states. Variable product terms, first introduced in bipolar PAL devices like the AmPAL22V10, help the device match the application. Programmable buried states use internal storage elements for state-machine design; until now, PAL devices lacked independent buried states. Such elements were used only in conjunction with I/O pins. Consequently, when an I/O pin was an input, its associated storage element was wasted.

At first, PAL devices were produced with fuse-based bipolar technology; they could not be reprogrammed. Next generation EPROM-based PLDs (or EPLDs) are reprogrammable, as are the still newer electrically erasable PLDs (E^2PLDs).

To be erased, EPLDs must be in expensive windowed ceramic packages through which they're exposed to ultraviolet light for a lengthy time. Easier to reprogram, E^2PLDs fit in inexpensive plastic packages.

Complete testability is another significant advantage of E^2 technology. Unlike bipolar and EPLDs, which require additional testing circuitry, E^2 devices use only the array and its operating circuitry to make DC, AC, and functional tests.

Finally, a PAL device must meet the specific demands of combinatorial, synchronous, and asynchronous applications. While some CMOS PLDs have attempted to address the broad range of applications with a "one-size-fits-all" approach, this has limitations. The device tends to have extra features that show up in overhead while not providing others that are key. The result is expensive and less than optimal.

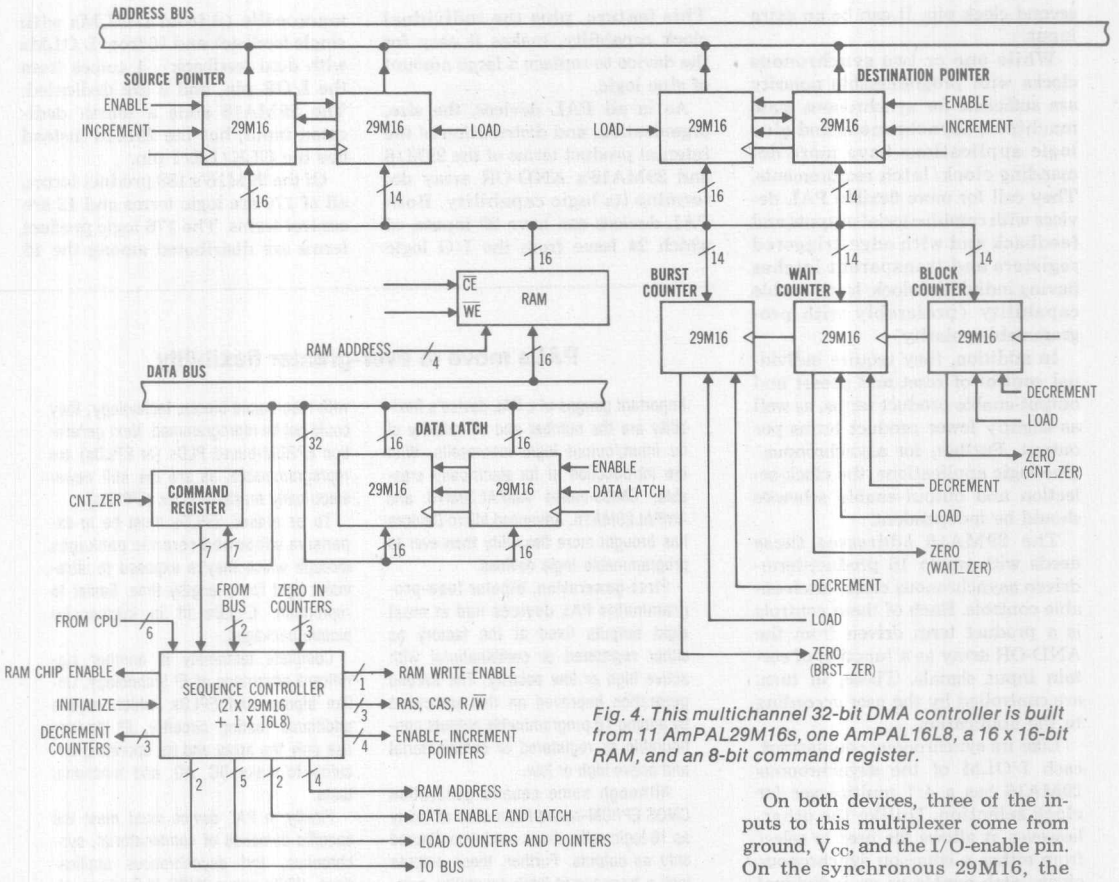


Fig. 4. This multichannel 32-bit DMA controller is built from 11 AmPAL29M16s, one AmPAL16L8, a 16 x 16-bit RAM, and an 8-bit command register.

I/OLMs, as was shown in Fig. 1. The eight I/OLMs with dual feedback have eight product terms each, while of those with single feedback, four have 12 product terms and four have 16. The average is 11 product terms per output.

This variable distribution of product terms, as well as the number of product terms per output, allows more complex functions—such as specialized counters and more complex state machines—to be implemented in a single device.

Of the 29M16's 12 control terms, two are used as common asynchronous reset and preset, and eight are output-enable product terms—one pair of common AND-XOR product terms per bank of four registers. The remaining two are used for observability in testing and for preload of the device's storage element.

As for the 29MA16, its smaller total of 178 product terms is divided into only 112 logic terms and all of 66 control terms. The logic product terms are also distributed in a variable fashion (see Fig. 2 again); the average is 7 product terms per output. Of the control terms, four are distributed to each of the 16 I/OLMs, and the remaining two are for observability and preload.

Output enable control

The I/O pin for both new devices may be configured as a dedicated (or permanently disabled) input, a dedicated output, or dynamically controlled. This flexibility results because the connection from the I/OLM to its I/O pin is controlled by a 4:1 multiplexer.

On both devices, three of the inputs to this multiplexer come from ground, V_{CC}, and the I/O-enable pin. On the synchronous 29M16, the fourth input to the multiplexer is driven by the two AND-XOR product terms already mentioned as being common for a bank of four I/OLMs. In synchronous state machines, this arrangement allows the I/O pins to be controlled in groups of four. The AND-XOR product term control also allows for programmable polarity, a useful feature for bus-control applications.

On the 29MA16, this fourth input is driven by an individual product term from the AND-OR array. Such individual product-term control gives users maximum flexibility in configuring the I/O pins.

Both the parts have built-in features to simplify testability and system-level debugging. These include a preload and a power-up reset for testability and an observability product term for debugging and for

tracing buried state registers.

When the observability product term is asserted, it suppresses the combinatorial output data, preventing the data from appearing on the I/O pin. It then presents the contents of the register/latches on the output pin for each of the logic macrocells.

The built-in power-up reset initializes the register/latches to a known state, a feature that simplifies state machine designs. A security cell in both devices protects the logic against unauthorized copying.

A multichannel DMA controller

The AmPAL29M16/29MA16 are suitable for applications ranging from glue and general-purpose logic replacement to synchronous and asynchronous state machines. Fairly complex Mealy and Moore state machines can be readily designed, and the I/O latches support double pipelined systems.

Figure 4 is the block diagram of a *direct memory access* (DMA) controller built for the 32-bit 68020 microprocessor. It is a simple, four-channel, single-cycle, 32-bit memory-to-memory data-transfer design.

The 68020 processor accommodates memory blocks and peripheral widths ranging from 8 to 32 bits on its system bus. It also imposes certain connectivity requirements on slave devices: 8-bit devices must go to the uppermost 8 bits of the data bus, while 16-bit devices must connect to the data bus's upper word.

Consequently, issues related to the data width for the DMA operation are the data-transfer size (8, 16, or 32 bits), the data-funneling scheme (for packing and unpacking data to support a flexible data-transfer scheme), and block size.

The DMA logic comprises two address pointers (one for source and one for destination); a 32-bit data latch; burst, wait, and block-length counters; an 8-bit command register; a 16 x 16-bit RAM; and the DMA sequence controller. This design uses a total of 11 CMOS AmPAL29M16s —9 to implement pointers, latch, and counters and the remaining couple in the sequence controller.

The two pointers have a width of 30 bits divided into an upper 16-bit segment register field and a lower 14-bit offset register field. The 16-segment register field allows data to be transferred in 64-K segments, each segment being a maximum of 16-K blocks (or 64-K words, with 4 words in each block).

The task of segment register maintenance comes under CPU control. Thus, if the data to be transferred spans a segment boundary, the CPU must break up the transfer into two separate operations and set up segment registers for it.

The 14-bit offset register's value is loaded into the RAM locations by the CPU before the DMA operation begins. In this example the DMA

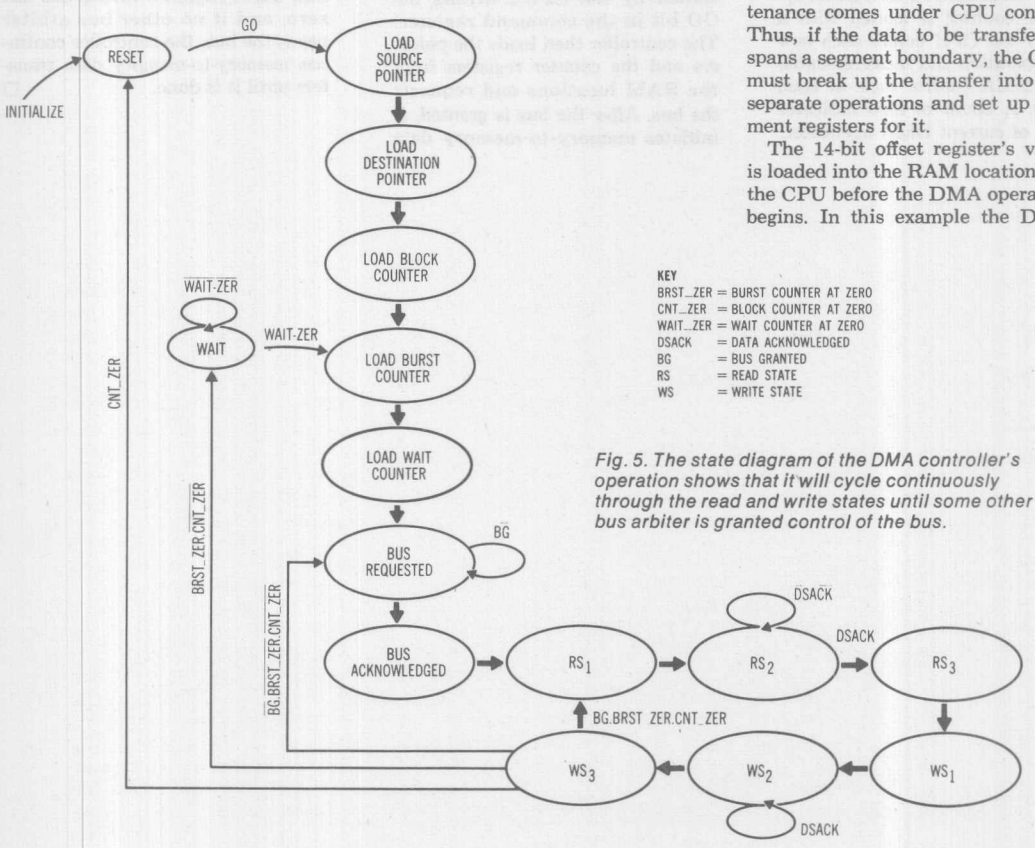


Fig. 5. The state diagram of the DMA controller's operation shows that it will cycle continuously through the read and write states until some other bus arbiter is granted control of the bus.

transfers only long words, so the least significant address bits A_1 and A_0 are stuck-at-zeroes. The offset register is incremented after every DMA read/write cycle unless the command register says not to.

A 32-bit data latch temporarily holds the long-word data fetched from the source. During destination writes, this becomes the source for the data. The burst, wait, and block count registers are each of them 14-bit decrementers.

The burst register counts down after every DMA read/write cycle. At zero, the data and address buses are relinquished by the DMA controller for a number of cycles defined in the wait register.

The 14-bit block counter decrementer, whose value is loaded into RAM locations by the CPU, keeps track of the number of long words transferred in each current DMA cycle. The counter is loaded with a value by the CPU before each new DMA operation and is decremented by the DMA control logic at each transfer. A count of zero indicates the end of current DMA operation.

The control sequencer

The DMA control sequencer has three jobs. It handles loading both the pointers and the registers from the RAM locations. It controls the DMA read/write cycles. And it controls the interfaces to the CPU (handling of burst and wait registers) and the memory.

Because of I/O limitations in the sequence, the buried registers associated with inputs implement the state sequence. The output-enable structure enhances the individual I/O control of every macrocell, and the clock scheme is used by triggering on both edges of the system clock to improve the DMA read/write cycle throughput.

According to the state diagram for the DMA sequence controller (see *Fig. 5*), the device stays in the idle state until a DMA operation is started by the CPU's writing the GO bit in the command register. The controller then loads the pointers and the counter registers from the RAM locations and requests the bus. After the bus is granted, it initiates memory-to-memory data transfers—read followed by write cycles. (One memory-to-memory transfer cycle is a single operation.)

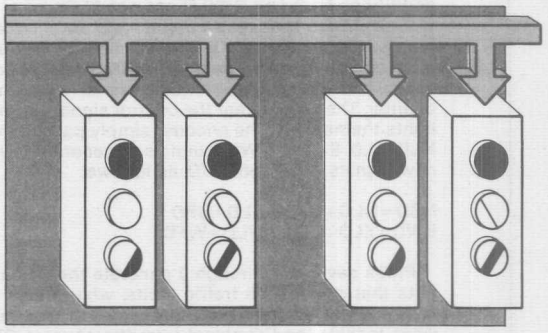
DMA preemption

The DMA preemption is performed in write state WS_3. When the external arbiter signals that the bus is needed by another potential bus master (by negating bus grant), the DMA controller gives up the bus and enters the bus-request state. It stays there until the bus control returns to the DMA controller.

In state WS_3, if the block counter reaches value 0, then it returns to the idle state. If the burst length has reached zero, but the counter value is not zero, then it goes to the wait state until the wait register value reaches zero. Then it transfers to the "load burst register" state. In state WS_3, however, if both the counter and burst-register values are not zero, and if no other bus arbiter needs the bus, the controller continues memory-to-memory data transfers until it is done. □

PROSE Devices Simplify State Machine Design

A bipolar device combines a PAL device for conditional branching with a PROM device for storing system control functions to make the design of small state machines easier.



Control logic is required in a broad range of applications—from low-level control of custom bit-slice microprocessors, to bus arbitration and timing generation in conventional microprocessors, to special-purpose needs like data encryption and decryption. Even though these applications vary in complexity, they all require a sequential-type device (a state machine, for example) that provides for sequences of output signals based upon state transitions and that includes some capability for branching. More complex sequencers provide capability for looping and subroutining.

Possible candidates for control logic range from Programmable Array Logic (PAL) devices, to the more complex Field Programmable Logic Sequencer (FPLS) devices, to instruction-based microprogram controllers that rely on on-chip memory for expanded subroutine capabilities. While FPLS and PAL devices have limited product terms for many state-machine applications, the other microprogram controllers are sometimes more complex than required. For applications requiring control functions that small and medium state machines can represent, the Prose device from Monolithic Memories Inc (MMI) can facilitate control logic.

Combining a PAL device for conditional branching and a PROM for storing the control functions of the system, Prose is intended to function as a programmable sequencer (Prose = PROgrammable SEquencer). It offers a maximum of 128 states and up to four-way branching. A bipolar device operating at up to 25 MHz, Prose functions as a Moore state machine. Under certain conditions, a Mealy state machine can also be modeled with Prose.

Nick Schmitz

Schmitz is a product planning manager for programmable logic devices at Monolithic Memories Inc (Santa Clara, CA).

Working faster and simpler

Custom microprocessors, such as those built with bit-slice components, require control logic in the form of multiple IC microinstruction memories and sequencers to generate addresses for those memories. For complicated sets of instructions with many states and complex branching and subroutines, a microprogram controller makes sense. But in situations in which a more limited set of instructions and instruction transitions is required, the single-chip Prose device is faster as well

Reprinted from the April 1, 1987 edition of Computer Design
Copyright 1987 by PennWell Publishing Company.

Prose in a classic traffic-controller application

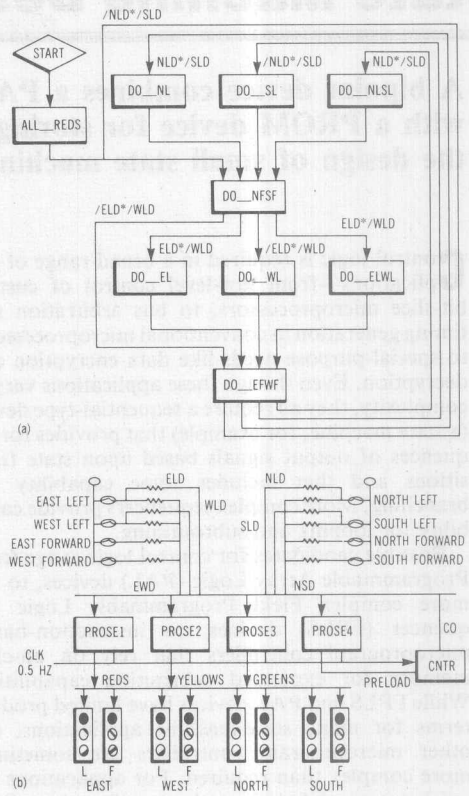
Consider a four-way intersection. For each direction (N, S, E and W), there are two sets of lights: one for the forward direction (F) and one for left turns (L). Each set has three lights: red (R), yellow (Y) and green (G). For each of the four directions, there are also two detectors (D): a left turn detector and a forward detector.

In the master diagram for traffic-flow control (a), each box represents a set of states generated by the controller in response to the input conditions. For instance, DO NL is executed given the input conditions NLD */SLD (NLD and not SLD).

Four Prose devices, a counter and an encoder are used to implement this controller (b). The inputs to this controller are the six outputs of the encoder and the carry out (CO) signal from the counter. The outputs are the control signals to the lights themselves. The encoder simply passes the NLD, SLD, ELD and WLD signals. It generates two new signals, NSD and EWD, as follows:

NSD = NLD + NFD + SLD + SFD
EWD = ELD + EFD + WLD + WFD

Prose devices 1 through 3 generate the 24 outputs that control the traffic lights, while Prose 4 preloads the counter. Note that all four Prose devices receive the CO signal from the counter. CO tells the Prose devices when the preloaded number of cycles has been completed so they can generate the next state. The counter is used to conserve states in the Prose devices; that is, the devices simply cycle back to the same state until the desired number of cycles has been completed or until some condition detected by the sense detectors warrants jumping out of the loop early.



as simpler. For instance, when an ALU is used only for a multiply or filtering operation, Prose devices reduce chip count but can also generate the required instructions faster than the more complex microprogram controller. Prose devices can also be used for addressing the data memory to fetch the operands for the filtering operation.

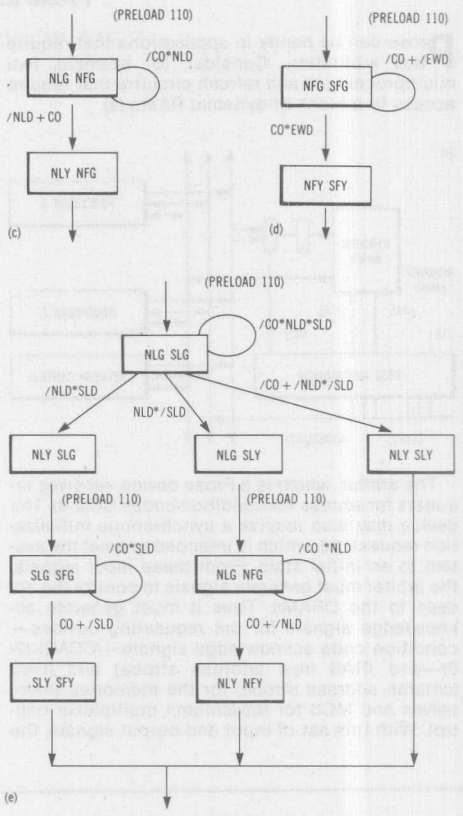
Consider also the case of a video controller. For scanning purposes, counters that operate in various sequences and count lengths are required. But instead of implementing these as actual counters, the sequences involved can be unlocked and implemented as state-machine transitions. And there's an advantage beyond the mere economy of parts. A count can be set or initiated and then left to poll various external hardware conditions while the microprocessor performs other operations.

Memory and bus arbitration applications offer further opportunities for using Prose devices. If two processors or other devices vie for memory access, a simple Prose state machine can implement the logic for deciding which device gets the bus and/or access to memory. In such a case, the heavy artillery of a microprogram controller and memory isn't required.

As a sequencer for signal-processing applications, Prose state machines offer speed and sufficient functionality without the overkill of a complex microsequencer paired with dedicated memory. For simple algorithms, such as those involved in performing a fast Fourier transform, Prose devices can control the set of vectors that are multiplied and added in the process. Via the instruction sets to the devices, Prose can control the operation

Consider a block such as DO NL. As we enter this block, we have just completed flow in the east-west direction. The previous state has generated the preload for the counter so that as we enter the block DO NL with NLD* /SLD, the counter begins the count. Suppose that we have preloaded 110 and count down. On the count following 000, CO goes high, and this tells the Prose devices to branch to another state. Figure (c) is a state transition diagram for DO NL. The state "NLG NFG" recycles until CO or /NLD occurs. Figure (d) shows the details of DO NLSL; Figure (e) shows DO NFSF. These diagrams reflect all basic transition types in the controller. There are four blocks of the type DO NL, two of the type DO NLSL, and two of the type DO NFSF. In addition, there's one other block, ALL REDS, that contains only one state: all outputs red. So, from counting the possible states of the system according to branch type, we have 15 one-way branches, 10 two-way branches, 0 three-way branches, and 4 four-way branches, resulting in 31 of the 128 PROM locations used.

This application uses four Prose devices—three for generating signals to control the lights and one for controlling the counter for state minimization. Although outputs are different, all four devices use the same inputs and execute the same transition table and thus can be considered to be in the same state, or to be generating a single state with a wider output word. This usage demonstrates how the Prose device can be expanded in width. When programming the devices, state transition and condition equations are identical and can be copied. Only the output equations are unique to each and must be programmed separately.



of the ALUs or multipliers and can control memory access when separate Prose devices are used. A Prose device is well suited to relatively simple operations that don't involve complicated decision making. For complex digital signal processing operations, a programmable DSP device may be better, but it's not likely to be as fast as the dedicated hardware approach.

In peripheral control, the simple state-machine approach can be efficient. Consider the case of run-length-limited code. Both encoding and decoding can be translated into state sequence diagrams, which examine the serial data stream as it's read, and can then be readily programmed into a Prose device.

Industrial control and robotics also require simple control functions. Tasks such as mechanical positioning of a robot arm, simple decision making, calculating a trigonometric function and displaying several digits usually don't require a high-power microsequencer with stacks and pointers and microprogram memory. What's required instead is a device that can store a limited number of states and allows simple branching upon conditions.

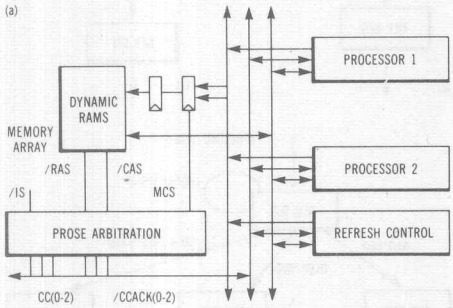
Control logic alternatives

A registered PAL device is a simple state machine and can be effectively used for simple control functions, such as counters, dynamic RAM controllers, interrupt controllers and certain types of video controllers. Combining a PAL device with memory such as a PROM, as MMI has done in Prose, gives the device much more powerful control capability.

Another alternative for control logic is the FPLS,

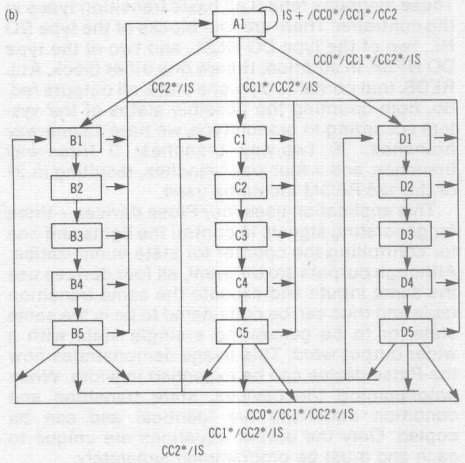
Prose as a bus arbiter

Prose can be handy in applications that require bus arbitration. Consider, for example, two microprocessors and refresh circuitry that require access to a block of dynamic RAMs (a).



The arbiter, which is a Prose device, receives requests for access via condition codes CC(2-0). The device may also receive a synchronous initialization request, /IS, which is intended to reset the system to an initial state. From these input signals, the arbiter must generate signals to control the access to the DRAMs. Thus it must generate acknowledge signals for the requesting devices—condition code acknowledge signals—/CCACK(2-0)—and /RAS (row address strobe) and /CAS (column address strobe) for the memories themselves and MCS for row/column multiplexer control. With this set of input and output signals, the Prose device can precisely control the access to memory.

Once access is granted, a complete memory cycle requires five clock cycles. Three clocks are required for access; two are required for DRAM precharging. /RAS is asserted from the second cycle to the end of the fourth, while /CAS is asserted from the third cycle to the end of the fourth. (Row and column addresses are asserted, as shown, prior to the assertion of /RAS and /CAS.) Both /RAS



which has programmable AND and OR arrays as well as JK or clocked-RS flip-flops with internal registered feedback that allow sequencing and configurable I/O pins. FPLS devices have a fixed number of product terms that can be shared between input logic, state transitions and output functions, dictated by the application needs. Because the Prose device has dedicated resources—the PAL and PROM arrays—these applications trade-offs aren't required. While not characteristic of the architecture, some FPLS sequencers operate at different speeds depending on the logic functions implemented; for example, as the complexity of the logic functions increases, speed decreases. With the Prose device, however, operation speed is a constant. For many applications, FPLS devices and Prose devices are complementary; a design engineer should be able to use both, choosing the right device for the application.

Finally, some recently introduced instruction-based machines, similar to a PROM under control of a microsequencer, offer another option for control logic. These machines are sophisticated devices with ample states for state machines, but they're too complex for simple control applications. Moreover, user adaptability and customization is limited. Because they're powerful and have sufficient outputs, these machines can easily implement counter-type state machines. But they offer a limited choice of combinations of input literals for condition testing. While such devices form a powerful microprogram controller/sequencer for generating addresses, they don't provide a single-chip solution, and in many cases they're far more powerful than the application requires.

Prose device overview

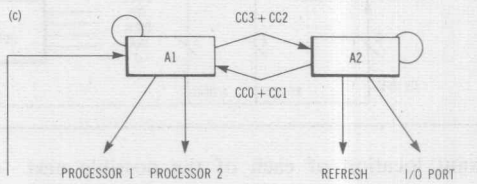
The major building blocks in the Prose device are the PAL array, the PROM array, and the shadow and output registers. In brief, the PAL array gener-

and /CAS are removed during the fifth and the first cycle for DRAM precharging.

The sequences of memory clock cycles with their associated control signals can be viewed as a set of state transitions (b). The condition codes, CC(2-0), are assigned as follows: CC2 is a refresh required; CC1 is a request from Processor 1; and CC0 is a request from Processor 2. A refresh cycle differs from a regular memory cycle only in that /CAS isn't asserted.

As shown in the transition diagram, A1 cycles upon itself if IS is H (or active low signal /IS is L), or if there's no request. If there's a refresh request and IS is L, the transition A1 to B1 occurs, initiating the cycle of states B1 through B5. This acknowledges the refresh request and generates the timing required to carry out the refresh operation. Note that any time /IS occurs, the next transition is to A1. The same is true for the loops C1 through C5 (processor 1) and D1 through D5 (processor 2). Note also that at B5 if there's no request (and IS is L), the next transition is to A1. But if IS is L and C2 is still true (another refresh request), the next transition will be to B1. Furthermore, if CC± is L (no refresh request), IS is L, but CC1 is H (processor 1 request), the transition will be to C1 directly. Similarly, if IS, CC2 and CC1 are all L, and CC0 is H, the transition will be directly to state D1. The transition diagram shows a similar relationship of transitions in the loops C1 through C5 and D1 through D5. There are 16 states, including 1 three-way, 12 two-way, and 3 four-way states, which the Prose device can easily accommodate.

Prose can also handle more complex configurations. Suppose there are four devices competing for bus and memory access. A problem arises in simply expanding the diagram (c) to accommodate the extra device. Doing this directly would require that A1 be a five-way branch. But the Prose is limited to four-way branching.



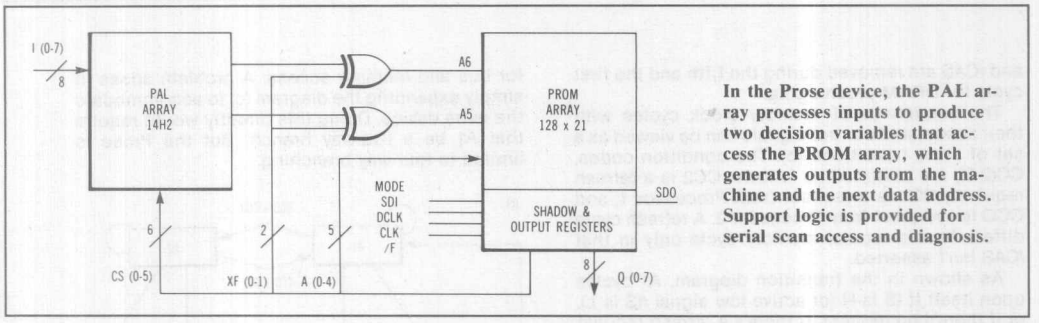
This difficulty is easily avoided, however, by inserting another state, A2, between A1 and the transition sequences for two of the requesting devices. Thus A2 serves as the embarkation point for these transitions. Here, the condition codes are CC3 (refresh), CC2 (processor 1), CC1 (processor 2) and the CC0 (an I/O port). CC3 has the highest priority; CC0 has the lowest. Note, however, that D1 through D5 are the five clock cycles for the refresh circuit. A refresh request always gets the highest priority but, since speed of execution isn't crucial for the refresh request, it's reached via A2. That is, the refresh request is never ignored, but it's adequate to service it through A2, allowing faster servicing (by one clock cycle) of processor 1 or processor 2 requests. A1 goes to A2 for CC3*/IS + CC0*/CC2*/CC1*/IS; A2 goes to D1 for CC3*/IS. This configuration requires no one-way, 16 two-way, 1 three-way, and 5 four-way states—easily handled by Prose.

ates the branch address bits for the memory (128 words×21 bits), which stores the system's next state addresses, device outputs and control functions. The shadow register is used for programming and diagnosis, and the output register provides synchronous operation and registered feedback.

The PAL array is a 14H2-type: it has 14 complementary inputs and two active high outputs. Each of the two outputs is a sum of eight product terms. Eight of the inputs, from an external source, are used for encoding of conditions for branch selection. The other six inputs, from internal feedback from the PROM, are used for selection of the conditions that determine branching when decisions are involved in state transitions.

The PROM consists of 128 21-bit words or locations, each of which can be viewed as a state, or as containing state-related information. Eight bits at each location are intended as outputs. The other 13 bits are intended as feedback: five are feedback to the PROM array itself for partially determining the next location to be addressed, and six are condition-select bits to the PAL array. They're used for product term selection/enabling. The remaining two bits are inputs to the exclusive OR gates. They're used to determine address bits A6 and A5 when branching isn't involved.

When no branching is involved, CS(5-0) select no condition in the PAL array, so that no product term is wasted, and the PAL outputs are X1=X2=0. Thus XF1 and XF0 are used to select the higher-order address bits—A6 and A5, in this case—and product terms are conserved for use when real conditional selection (branching) is involved. When conditional branching is involved, no more than four branches can be from one location, and all branch states will be in the same primary location but in different quadrants. That is, address bit A(4-0) will be the same, and the conditions will toggle A6 and A5 to determine the exact



In the Prose device, the PAL array processes inputs to produce two decision variables that access the PROM array, which generates outputs from the machine and the next data address. Support logic is provided for serial scan access and diagnosis.

quadrant location of each of the possible next states.

For diagnostic purposes, a shadow register breaks the data path. Data can be shifted in or out serially. The shadow register and the output register are operated by different clocks, so that data may be shifted without affecting device operation and spoiling data. This lets data (in this case, test vectors) be shifted into the shadow register and clocked onto the output register. A normal operation can then be performed, after which the data can be loaded onto the shadow register and shifted out to be analyzed. In addition, a multiplexer to the shadow register lets the register contain data from the output register or from the previous shadow register contents.

Internally, programming of the Prose device takes place via the shadow register, is completely transparent to the user and is done individually for each half of the device. For programming the PAL part of the device, a certain pattern of data is shifted into the shadow register with V_{CC} raised to the programming supervoltage and the preset/enable pin also raised to the supervoltage afterwards. Individual fuses in the PAL device are blown and selected by the appropriate pattern.

Programming of the PROM is accomplished in similar fashion. V_{CC} is raised to the supervoltage value; SDI is raised to the supervoltage afterwards. The bit pattern in the shadow register determines which fuse is to be blown.

Software support

Prose state machines are fully supported by new programs in the Palasm suite—Proasm for assembly and Prosim for simulation. To program the Prose device, the user inputs a list of state names, transitions and conditions that cause transitions to subsequent states. When progamming a Moore machine, the user also inputs a list of expected outputs for each state, and when programming a Mealy machine, the user inputs a list of outputs expected for each path to each state. (For a Moore state machine, the outputs depend only on the state. For a Mealy machine, outputs depend on the inputs also.)

User input to Palasm 2 (for state-machine entry) consists of four sections: declaration, state, equations and simulation. The declaration section specifies the type of Prose device to be programmed. Currently, there's just one device in the family.

The state section specifies the state transitions and the output details. The user specifies the type of state machine—Moore or Mealy—and the various defaults that should be used when the next state or the outputs can't be determined from the equations. Furthermore, the user specifies state transitions and outputs via equations. The transition equations specify transitions from each current state. The output equations specify the outputs for each state, in the case of a Moore state machine, or the outputs expected upon the transition from one state to another, in the case of a Mealy state machine. For the Moore machine, there's only one set of outputs for each state, but for the Mealy machine, the outputs depend on the path to the state. The last section, the simulation section, contains instructions that tell the software how to simulate actual operation of the state machine and, therefore, how to test the device once it's programmed. **CD**

Designing a state machine with a programmable sequencer

Programmable-array-logic and PROM combo provides a convenient approach to state-machine design for local intelligent control

Frank Lee, Product Planning and Applications
Monolithic Memories, Inc., Santa Clara, CA

In the abstract, a computer, or at least parts of a computer, can be defined by a state-machine directed-transition graph (see *Fig. 1*). A state machine is a small, locally controlled machine—not intended for a complete computation of a complex program.

Experts generally divide state machines into two basic classifications: algorithmic and synchronous-sequential machines. Algorithmic machines include the subcategories called Turing machines, Post machines, and finite machines, which primarily describe the specific algorithm used in programming the computer. A basic difference between a synchronous sequential state machine and an algorithmic machine is the presence or absence of stop states. A start state, of course, must be present in all machines or else they will not be initiated.

A stop state in an algorithmic machine (like a Turing) occurs when it has finished executing an algorithm. In a synchronous sequential machine, however, to achieve a stop state, the machine must branch to itself—a type of wait state—until another operation is ready. The only time a synchronous sequential state machine may be actually halted is for an error or illegal command that forces it into looping on itself unconditionally. But once in such a state, the state machine theoretically can't exit from it. The machine must be reset and started over again.

Implementing a state machine

Although devices such as PAL, PLA, and FPLS arrays can be used to implement such a state machine, a *programmable sequencer* (PROSE) device, designated PMS14R21, is a convenient, flexible, and economical way. It consists of a combination of a special PAL and a registered PROM that are optimized for use in local, intelligent-control state-

Reprinted from ELECTRONIC PRODUCTS Magazine—February 1, 1987., AR-160
645 Stewart Ave., Garden City, NY 11530 • © 1987 Hearst Business Communications, Inc.

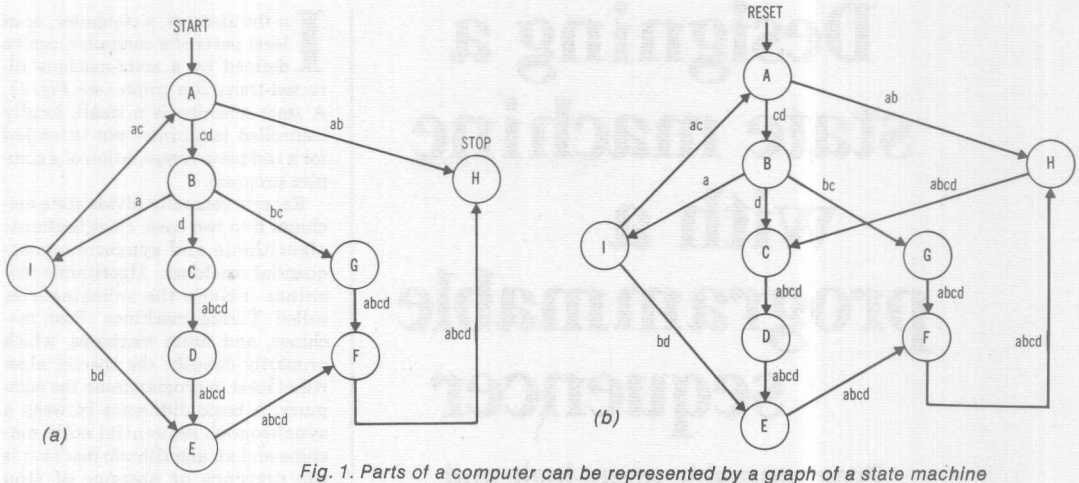


Fig. 1. Parts of a computer can be represented by a graph of a state machine showing its directed transitions between states. An algorithmic machine (a) has stop state(s), whereas a synchronous machine (b) does not.

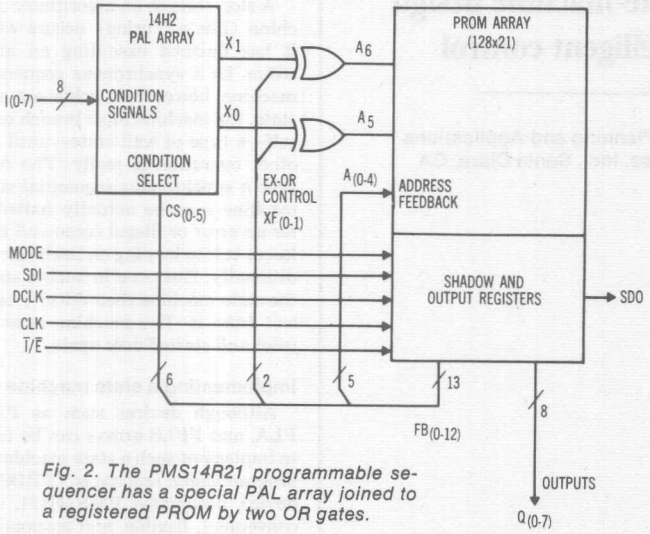


Fig. 2. The PMS14R21 programmable sequencer has a special PAL array joined to a registered PROM by two OR gates.

machine design (see *Fig. 2*).

The introduction of the PROSE device is a significant advance in the design of fuse-programmable state machines of small to middling size. The PAL array serves to select transition conditions, and the PROM stores the state assignments. Also, two exclusive-OR gates help reduce the total number of states and product terms a state machine would need. One terminal (I/E) can be programmed to receive an asynchronous initialization signal (one that resets all registers to the high state) or an output enable (via an architectural fuse); others, to enable fault diagnosis. A power-up reset feature is also provided.

Each PROSE device has eight inputs, eight outputs, up to 128 states, and up to a four-way branching capability. Many local control functions can be done faster than on a good many other state machines. Also, timing restraints on external devices are greatly relaxed. Perhaps most important of all, the PROSE device provides a small, single-package solution to state-machine design (for example, 24-pin, 3-in. Skinny-DIPs and 28-pin plastic leaded chip carriers).

Moore and Mealy models

Two categories of state-machine models are generally implemented: the Moore and the Mealy state machines. The Moore machine's outputs depend only on the state; they are independent of inputs. A Mealy machine's outputs, however, depend on both the state and the machine's inputs. The Mealy, therefore, is a superset of a Moore state machine.

A true Mealy state machine is too versatile to be defined in a general architecture. A Moore machine is much easier to define and simpler to implement. The simpler the state machine is, the faster it will be. On the other hand, for most applications, the outputs of a state machine depend somehow on the inputs as well as on the state.

The PROSE device can support the Moore model. Fortunately, the Mealy model can also be supported, provided that the PROSE device's output pattern can be pipelined by one clock cycle. Also, the number of

combinations of patterns in the subsequent states, and the state thus pipelined, must not exceed four.

Controlling a state machine

Naturally, a state machine must include a control system in addition to the operating portion outlined so far. The control system determines the next state on the basis of the current state plus a combination of inputs to the control system. Of course, based on its state, the operating part of the system determines the outputs of the machine.

The number of inputs, to a certain extent, determines the number of conditions (as inputs or combinations of input literals) from which the state machine can select. It can also determine whether encoding the conditions is necessary, which then becomes part of any external delay. A small, local, intelligent, arbitration unit may have as few as three to four states to as many as 30 to 40 states. A highly intelligent controller can have thousands of states. However, a state machine generally has a relatively small number of states to which branching occurs, because the latter states will eventually loop to each other (except maybe for the few starting states).

Conditional branching needed

Because a state machine switches its states on the basis of both the current condition and inputs, frequently conditional branching is necessary, sometimes to more than two next states. Four-way branching is common. If more than four-way branching is required, up to 16-way branching can be carried out by two cycles of four-way branches (see *Fig. 3*).

But in applications that operate on single-cycle instructions, such a branch-fanout tree may not be desirable. Such single-cycle multiple branching can be done provided the machine allows losing a cycle for the fanout and has enough product terms to perform the branching on all the conditions. (A PAL or PLA device has a limited number of available product terms.)

Similarly, the number of outputs determines whether encoding of operations control is necessary and how many operations are possible. Again, decoding of control signals externally will mean extra delay. In a single-device state machine, since a limited number of pins is available, the ratio of the number of outputs to the number of inputs is very critical. This ratio determines the application range of the device.

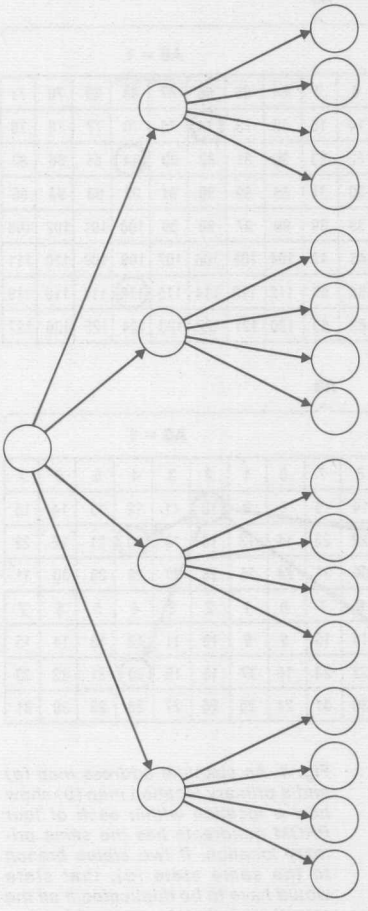


Fig. 3. A state fanout tree of up to a 16-way branch can be carried out in two cycles of four-way branches. This operation requires the state machine to lose a cycle for the fanout and to have enough product terms available to perform the branching for all the conditions.

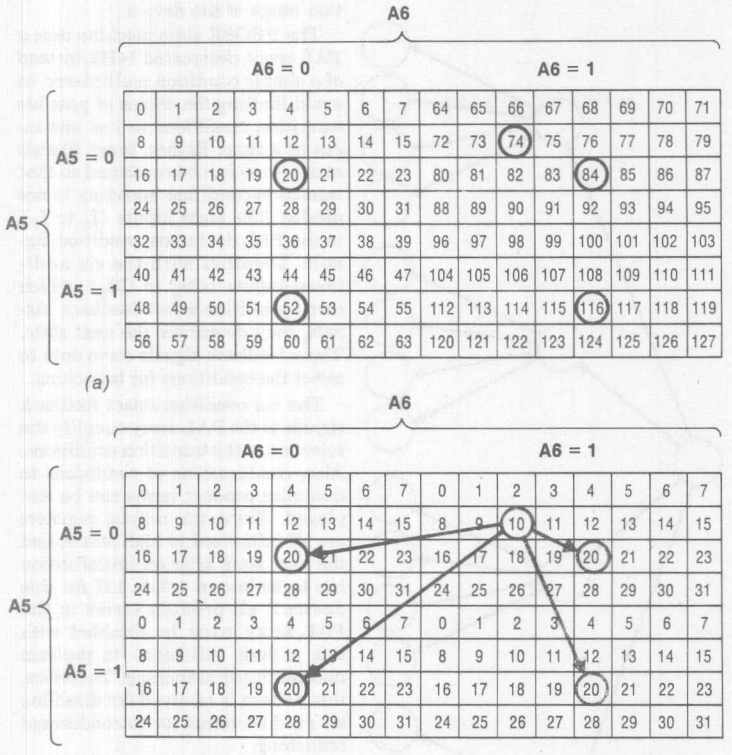
The PROSE state machine uses a PAL array, designated 14H2, instead of a simple condition multiplexer, to avoid limiting the choice of possible transition conditions to just one input at a time. Rather, input literals at any time can be combined so that transition-condition encoding is not needed. The eight inputs (I_0 to I_7) to the PAL device are condition signals. Together with the six additional inputs (CS_0 to CS_5), which carry condition-select feedback signals, they determine the next state. These feedback signals serve only to select the conditions for branching.

The six condition-select feedback signals to the PAL array simplify the selection of the transition conditions. Also, combinations of conditions to minimize product terms can be employed. Since the output registers are all initialized to high states, and the next state after an initialization has to be known (state 127 for this device), all product terms in the PAL array must be disabled with this pattern (all high) to perform unconditional branching. Therefore, this pattern is reserved for disabling all product terms for unconditional branching.

The 14 PAL inputs are complemented internally to help program two eight-product-term OR gates at each of two active-high outputs (X_0 and X_1). The conditions controlling the next states must fit within these limits. The outputs of the PAL feed *exclusive-OR* (EXOR) gates, which help conserve the need for state and product terms.

To perform an unconditional, or one-way branch, to save a product term, X_0 and X_1 are at 00. For example, the new PROM address has A_5 and A_6 equal to 10 (binary). Accordingly, the EXOR feedback signals XF_0 and XF_1 are programmed as 10_2 to force the EXOR outputs to 10 unconditionally.

For initialization, the CS_0 to CS_5 feedback pattern for an unconditional branch is 111111. Thus, the operational feedback of CS_0 to CS_5 to the PAL must contain at least one 0—that is, patterns like 111111, 11X1X1, or XX1XXX should not



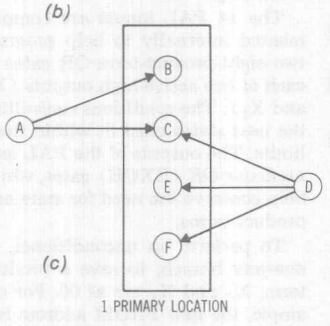


Fig. 4. An absolute address map (a) and a primary location map (b) show how a location within each of four PROM quadrants has the same primary location. If two states branch to the same state (c), that state would have to be duplicated if all the resulting next states exceed four.

be used to select any product term.

Organization of the PROM array

The PROM array is a fuse-programmable array of 128 locations by 21 outputs. Eight primary outputs are directly available (Q_0 to Q_7). The actual functions are defined by the user. In addition, 13 more outputs serve as feedback signals to help determine the next state. All the outputs are registered at the rising edge of the CLK input signal.

The primary locations in the PROM are addressed by five inputs, A_0 to A_4, which are fed back directly from the PROM registered outputs to the PROM array. These 5 bits determine the next primary location.

Therefore, 32 primary locations define the states. The remaining two address inputs to the PROM (bits A_5 and A_6), which the PAL array and the exclusive-OR gates generate, address four PROM quadrants, in each of which branching can be performed to any of its 32 primary locations (see *Fig. 4a*).

Potentially, each of the 128 locations thus addressed could define a different state. In actuality, the number implemented is usually fewer. For example, some states may need to be duplicated in different primary locations. Consider a state A whose next states will be B, C, and D, and a state E whose next states will be D, F, and G. Although one next state, D, is common to the two conditional transfers, putting B, C, D, F, G within the same primary location (within the four quadrants) is impossible, since a primary location can accommodate only four next states in all (see *Fig. 4b*). Therefore, two primary locations must be occupied with the state D and its clone.

Another example requiring duplication of states occurs even when a state A branches to states B and C, and state D branches to states C, E, and F—state C being common (see *Fig. 4c*). Although the total number of next states is four, if insufficient product terms are available, the states of A and D may still need to be duplicated to achieve that result.

In the worst case, if every next state had to be duplicated because of incompatible combinations, or if all states had four-way branches, the PROSE state machine could still accommodate 32 states. Each state would have to be pointed to a different primary location.

Registers for the PROM

To conserve terminal pins, the 13 bits of feedback signals from the 21-bit output register are buried within the PROSE chip (see *Fig. 5*). What is called a shadow register, together with a *diagnostic-on-chip* (DOC) scheme, enables testing various parts of the chip, including the buried-signal conditions.

The shadow and output registers (see *Fig. 6a*) are operated by different clocks—DCLK and CLK, respectively—in several modes with the MODE and SDI signals (see *Table 1*). The output registers can multiplex data from the shadow register (to receive a test vector) or from the PROM array (to receive a test result). Similarly, the shadow register can multiplex data from the

output register (for the test result); the previous shadow-register bits (to shift in a test vector and/or shift out a test result); and from the current shadow-register bits (as just a hold operation).

The DOC-register concept also solves the programming issues for the PROSE device. A pattern in the shadow register identifies the fuse to be blown (see *Fig. 6b*). Bits PT_0 to PT_2 select one product term out of eight from each of the PAL's sum of terms. Of these, bits CS_0 to CS_5 and I_0 to I_7 select one input to the PAL array. The bit labeled O/P selects a sum term, and the bit T/C selects either the true or complementary value of the input selected.

Similarly, the fuse to be blown in the PROM is identified by the pattern in the shadow register (see *Fig. 6c*). A_0 to A_6 is the absolute address for the PROM array, and OA_0 to OA_4 selects 1 bit out of the 21 bits in a PROM word.

Applications abound

The PROSE device is designed to be used as a general-purpose state machine for controlling equipment or for data manipulation. The 128 states and four-way branch capabilities make it possible to be used as a high-speed memory or bus arbitrator, a video-RAM arbitrator, an industrial traffic controller, a signal-processor sequencer, a sequencer for mass storage or communications, a data encoder and decoder, and for many other applications. For example, the PROSE state machine can implement the *run-length-limited*

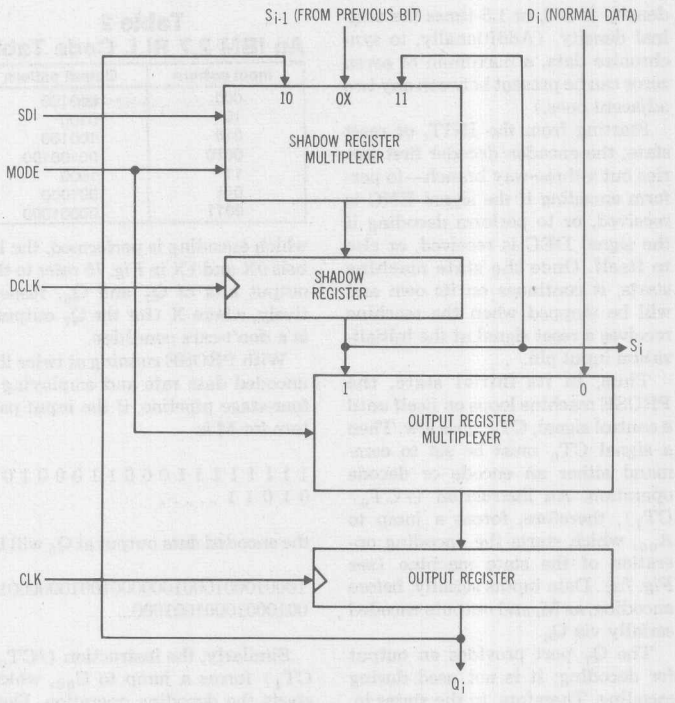


Fig. 5. The block diagram for a diagnostic register bit includes both a shadow register and an output register. Thus a test vector and the test result can be shifted in and out of the shadow register without corrupting the output flow.

code, designated 2,7 RLL (see *Table 2*). It is popular in mass-storage equipment and can increase the capacity of such a device by approximately 50%.

In any mass-storage media, a minimum separation of at least one zero between adjacent logic ones is required. In the 2,7 RLL coding/decoding system, at least two zeros must exist between any two adjacent ones. (Thus encoded patterns like 11 or 101 are illegal because there are fewer than two zeros between adjacent ones.)

Since the distance between ones determines the flux transition frequency on the magnetic media, a minimum of two zeros between adjacent ones means that flux transitions are at least three bit intervals apart. Therefore, the encoded flux density becomes 3/X (a binary 100 in the space X, the minimum on the medium between transitions). Other schemes like MFM, however, require a density of 2/X (a binary 10 in the minimum space X). Accordingly, since three encoded bits occupy the space on the media that carried 2 bits before encoding, the encoded

Table 1. Functions of the Diagnostic Registers

MODE	SDI	CLK	DCLK	Shadow register (SREG)	Output register (OREG)	Operation
L	X	C	*	hold	$OREG_i \leftarrow PROM$	normal
H	X	C	*	hold	$OREG_i \leftarrow SREG_i$	load from SREG
L	X	*	C	$SREG_i \leftarrow SREG_{i-1}$ $SREG_0 \leftarrow SDI$	hold	shift SREG
H	L	*	C	$SREG_i \leftarrow OREG_i$	hold	load to SREG
H	H	*	C	hold	hold	no operation

L = zero, H = one, X = don't care
C = clock pulse (low-to-high transition), * = stable clock or high-to-low transition, i = bit number

density is 3/2, or 1.5-times the original density. (Additionally, to synchronize data, a maximum of seven zeros can be present between any two adjacent ones.)

Starting from the INIT, or reset state, the encoder/decoder first carries out a three-way branch—to perform encoding if the signal ENC is received, or to perform decoding if the signal DEC is received, or else to itself. Once the state machine starts, it continues on its own and will be stopped when the machine receives a reset signal at the initialization input pin.

Thus, in its initial state, the PROSE machine loops on itself until a control signal, CT_1, goes low. Then a signal CT_0 must be set to command either an encode or decode operation. An instruction ($/CT_0/CT_1$), therefore, forces a jump to A_{00}, which starts the encoding operation of the state machine (see *Fig. 7a*). Data inputs serially, before encoding, as M, and outputs encoded serially via Q_6.

The Q_7 port provides an output for decoding; it is not used during encoding. Therefore, in the states in which encoding is performed, the labels 0X and 1X in *Fig. 7b* refer to the output bits of Q_7 and Q_6, respectively, where X (for the Q_7 output) is a don't-care condition.

Table 2
An IBM 2,7 RLL Code Table

Input pattern	Output pattern
000	000100
10	0100
010	100100
0010	00100100
11	1000
011	001000
0011	00001000

With PROSE running at twice the uncoded data rate and employing a four-stage pipeline, if the input pattern for M is

1 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 0 1
0 1 0 1 1 . . . . ,

the encoded data output at Q_6 will be

1000100010001000000100100000010
00100010001001000...

Similarly, the instruction ($/CT_0/CT_1$) forces a jump to C_{00}, which starts the decoding operation. During decoding, Q_6 is not used, but now Q_7 is the decoded serial data output. Therefore, the labels for the output pattern of Q are either 0X or 1X, where again X stands for a don't care.

For example, if the input pattern before decoding is the serial data input N:

1000100010001000000100100000010
0010001000100...,

the output pattern at Q_7 will be

1111111000110001010....

with each bit appearing twice, since all legal data is clocked twice.

In a practical design, if erratic patterns occur in the decoded data stream, the encoder/decoder should neither correct the data nor stop, but should make an intelligent guess at the error's probable nature and continue without holding up the data stream. However, in this example, illegal patterns have been made to enter an error state, because this situation is much easier to follow for illustration purposes.

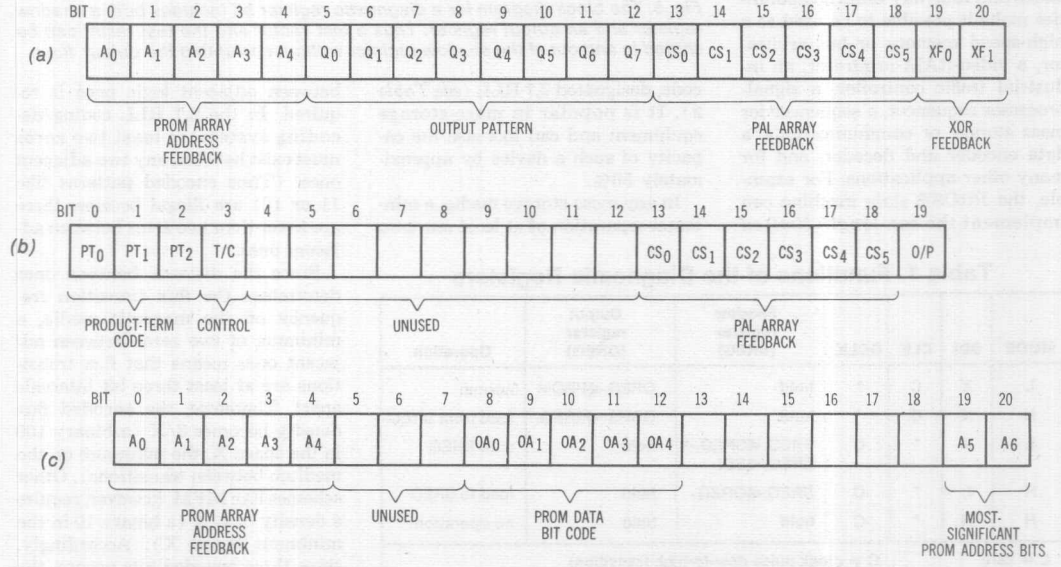
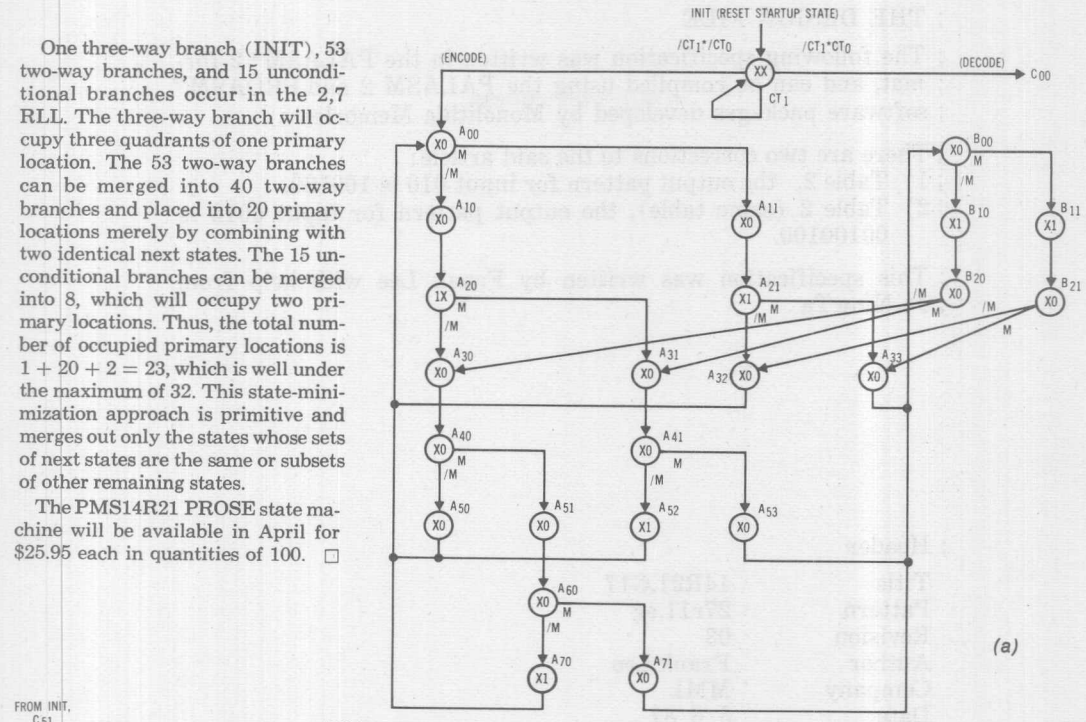
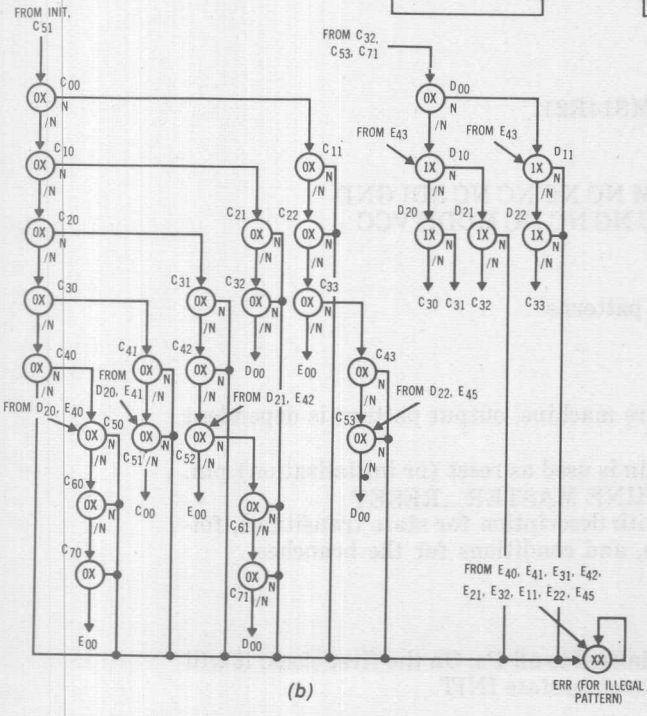


Fig. 6. Bit assignments for the shadow and output registers during normal operation (a) are quite different from the shadow and the output registers during PAL array programming (b) and from the shadow and the output registers during PROM array programming (c).

One three-way branch (INIT), 53 two-way branches, and 15 unconditional branches occur in the 2,7 RLL. The three-way branch will occupy three quadrants of one primary location. The 53 two-way branches can be merged into 40 two-way branches and placed into 20 primary locations merely by combining with two identical next states. The 15 unconditional branches can be merged into 8, which will occupy two primary locations. Thus, the total number of occupied primary locations is $1 + 20 + 2 = 23$, which is well under the maximum of 32. This state-minimization approach is primitive and merges out only the states whose sets of next states are the same or subsets of other remaining states.

The PMS14R21 PROSE state machine will be available in April for $25.95 each in quantities of 100. □





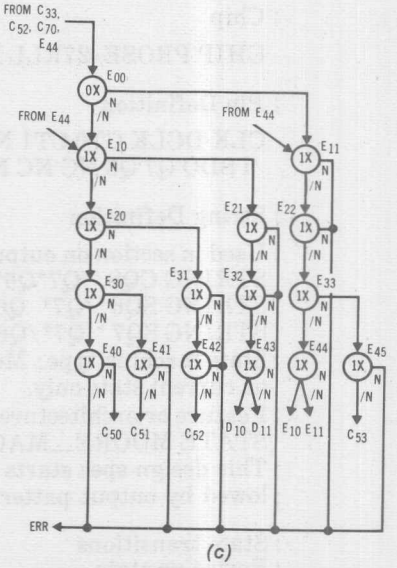


Fig. 7. The state-machine diagrams of a 2,7 RLL encoder (a) and its decoder (b) are a typical example of the possible application of the PMS14R21 device.

```
; THE DESIGN FILE
; The following specification was written in the PALASM® 2 for-
; mat, and can be compiled using the PALASM 2 and PROASM™
; software packages developed by Monolithic Memories.

; There are two corrections to the said article:
; 1) Table 2,  the output pattern for input 010 is 100100.
; 2) Table 2 (same table), the output pattern for input 0010 is
     00100100.

; This specification was written by Frank Lee with help from
; Jo-Ning Ta



; Header

Title          14R21.G17
Pattern        27r11.eg
Revision       03
Author         Frank Lee
Company        MMI
Date           3/6/87

; Chip

CHIP PROSE-27RLL PMS14R21

; Pin Definition

CLK DCLK CT0 CT1 N M NC NC NC NC SDI GND
/I SDO Q7 Q6 NC NC NC NC NC NC MODE VCC

; String Definition

; Used in section on output patterns
STRING CQS '/Q7*Q6'
STRING SQ6 '/Q7*  Q6'
STRING SQ7 '  Q7*/Q6'
; State machine type: Moore machine, output pattern is dependent
; on current state only.
; Feature or architecture pin is used as reset (or initialization) pin.
STATE MOORE_MACHINE MASTER_RESET
; This design spec starts with description for state transitions, fol-
; lowed by output patterns, and conditions for the branches.

; State transitions
; Power-up state

; The PROSE device is initialized to all 1's. On the first clock, it will
; perform unconditional branch to state INIT
```

```
POWER UP: = VCC -> INIT

; The PROSE will stay at INIT until there is a command for it to
; start encoding (condition ENC) or decoding (DEC). A00 is the
; starting state for encoding and C00 is the starting state for de-
; coding.
  INIT := ENC -> A00 + DEC -> C00 + -> INIT

; Encoder design

; A00,A10,...,B00,... are states for the encoder, as defined in fig. 7.
; EIN is defined in the condition section. It is the same as input M.
; VCC is an always-true condition.
; A00 will branch to A11 on EIN, else to A10
  A00 := EIN -> A11 + -> A10
; A10 : always branches to A20
  A10 := VCC -> A20
  A11 := VCC -> A21
  A20 := EIN -> A31 + -> A30
  A21 := EIN -> A33 + -> A32
  A30 := VCC -> A40
  A31 := VCC -> A41
  A32 := VCC -> A00
  A33 := VCC -> B00
  A40 := EIN -> A51 + -> A50
  A41 := EIN -> A53 + -> A52
  A50 := VCC -> A00
  A51 := VCC -> A60
  A52 := VCC -> A00
  A53 := VCC -> B00
  A60 := EIN -> A71 + -> A70
  A70 := VCC -> A20
  A71 := VCC -> A20
  B00 := EIN -> B11 + -> B10
  B10 := VCC -> B20
  B11 := VCC -> B21
  B20 := EIN -> A31 + -> A30
  B21 := EIN -> A33 + -> A32

; Decoder design

; C00,C10,...,D00,...,E00,... are states for the decoder, as defined in
; fig. 7.
; DIN is defined in the condition section. It is the same as input N.
; C00 will branch to C11 on DIN, else to C10
  C00 := DIN -> C11 + -> C10
  C10 := DIN -> C21 + -> C20
; ERR is the error state for an illegal encoded pattern received.

  C11 := DIN -> ERR + -> C22
  C20 := DIN -> C31 + -> C30
  C21 := DIN -> ERR + -> C32
  C22 := DIN -> ERR + -> C33
```

```
C30 := DIN -> C41 +-> C40
C31 := DIN -> ERR +-> C42
C32 := DIN -> ERR +-> D00
C33 := DIN -> C43 +-> E00
C40 := DIN -> C50 +-> ERR
C41 := DIN -> ERR +-> C51
C42 := DIN -> ERR +-> C52
C43 := DIN -> ERR +-> C53
C50 := DIN -> ERR +-> C60
C51 := DIN -> ERR +-> C00
C52 := DIN -> C61 +-> E00
C53 := DIN -> ERR +-> D00
C60 := DIN -> ERR +-> C70
C61 := DIN -> ERR +-> C71
C70 := DIN -> ERR +-> E00
C71 := DIN -> ERR +-> D00
D00 := DIN -> D11 +-> D10
D10 := DIN -> D21 +-> D20
D11 := DIN -> ERR +-> D22
D20 := DIN -> C31 +-> C30
D21 := DIN -> ERR +-> C32
D22 := DIN -> ERR +-> C33
E00 := DIN -> E11 +-> E10
E10 := DIN -> E21 +-> E20
E11 := DIN -> ERR +-> E22
E20 := DIN -> E31 +-> E30
E21 := DIN -> ERR +-> E32
E22 := DIN -> ERR +-> E33
E30 := DIN -> E41 +-> E40
E31 := DIN -> ERR +-> E42
E32 := DIN -> ERR +-> E43
E33 := DIN -> E45 +-> E44
E40 := DIN -> C50 +-> ERR
E41 := DIN -> ERR +-> C51
E42 := DIN -> ERR +-> C52
E43 := DIN -> D11 +-> D10
E44 := DIN -> E11 +-> E10
E45 := DIN -> ERR +-> C53

; Error stay until initialized
  ERR := VCC -> ERR            ;STAY UNTIL RESET

; <state>.OUTF refers to the output pattern for the <state>.
; CQS, SQ6, and SQ7 are output patterns. CQS means output bits
; 6 and 7 are all 0's. SQ6 means output bit 6 is 1 and output bit
; 7 is 0. SQ7 means output bit 7 is 1 and output 6 is 0. In all
; cases, output bits 0-5 are all don't cares as far as this design
; is concerned.
```

```
INIT.OUTF = CQS
A00.OUTF := CQS
A10.OUTF := CQS
A11.OUTF := CQS
A20.OUTF := SQ6
A21.OUTF := SQ6
A30.OUTF := CQS
A31.OUTF := CQS
A32.OUTF := CQS
A33.OUTF := CQS
A40.OUTF := CQS
A41.OUTF := CQS
A50.OUTF := CQS
A51.OUTF := CQS
A52.OUTF := SQ6
A53.OUTF := CQS
A60.OUTF := CQS
A70.OUTF := SQ6
A71.OUTF := CQS
B00.OUTF := CQS
B10.OUTF := SQ6
B11.OUTF := SQ6
B20.OUTF := CQS
B21.OUTF := CQS
C00.OUTF := CQS
C10.OUTF := CQS
C11.OUTF := CQS
C20.OUTF := CQS
C21.OUTF := CQS
C22.OUTF := CQS
C30.OUTF := CQS
C31.OUTF := CQS
C32.OUTF := CQS
C33.OUTF := CQS
C40.OUTF := CQS
C41.OUTF := CQS
C42.OUTF := CQS
C43 OUTF := CQS
C50.OUTF := CQS
C51.OUTF := CQS
C52.OUTF := CQS
C53.OUTF := CQS
C60.OUTF := CQS
C61.OUTF := CQS
C70.OUTF := CQS
C71.OUTF := CQS
D00.OUTF: = CQS
D10.OUTF: = SQ7
D11.OUTF: = SQ7
D20.OUTF: = SQ7
D21.OUTF: = SQ7
D22.OUTF: = SQ7
```

```
E00.OUTF := CQS
E10.OUTF := SQ7
E11.OUTF := SQ7
E20.OUTF := SQ7
E21.OUTF := SQ7
E22.OUTF := SQ7
E30.OUTF := SQ7
E31.OUTF := SQ7
E32.OUTF := SQ7
E33.OUTF := SQ7
E40.OUTF := SQ7
E41.OUTF := SQ7
E42.OUTF := SQ7
E43.OUTF := SQ7
E44.OUTF := SQ7
E45.OUTF := SQ7
ERR.OUTF := CQS


CONDITIONS
; The conditions for state transitions
ENC = /CT1*/CT0    ; encode command
DEC = /CT1* CT0    ; decode command
EIN = M            ; encoder input (original NRZ data)
DIN = N            ; decoder input (encoded data)


; SIMULATION
; Simulation vectors not shown here
```

FPCs and PLDs simplify VME Bus control

Designer's Guide to VME Bus Control—Part 1

By using fuse-programmable controllers (FPCs) and PLDs, you can implement VME Bus control in your computer system with a minimum of hardware. Just a few chips per plug-in module, and a bus arbiter, can perform all bus-control functions. This article, Part 1 of a 2-part series, describes the bus-arbitration process and control functions and shows you how to implement the VME Bus protocol in an FPC. Part 2 will describe the implementation of slave controllers, discuss interrupt handling, and provide tools for programming the FPC.

Arthur Khu, *Advanced Micro Devices*

Until recently, you needed many ICs—sequencers, microprogrammed control stores, and other MSI/LSI chips—to implement VME Bus control in a computer system. Now, however, you can use a minimum of hardware to handle the bus's intermodule communication functions. Just a few fuse-programmable controllers (FPCs) and programmable-logic devices (PLDs) relieve the CPU of all bus-control functions.

A typical VME Bus-based computer system comprises one or more master modules (eg, CPU boards), one or more slave modules (eg, cache/memory boards), a bus arbiter, and interrupt-handling circuitry. A master initiates a data transfer to a slave by requesting control of the data bus from the bus arbiter. Once bus control has been granted to a master, the master and slave exchange control signals according to predefined protocols that guarantee an orderly transfer of data between the communicating modules. The interrupt-handling circuitry services all interrupt requests.

By using the streamlined design in Fig 1, you can implement most intermodule communication in a VME Bus-based system by using just two types of VLSI devices: the Am29PL141 fuse-programmable device and the AmPAL22V10 programmable-logic device. For the remainder of the bus-control functions, you'll need some bus-arbitration circuitry, which must occupy a particular slot on the VME Bus backplane, but which can reside on the same board with the bus master of highest priority.

When you design VME Bus control into your system,

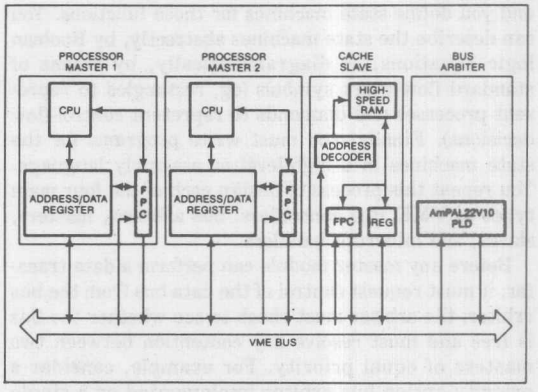


Fig 1—You can implement VME Bus control *in a computer system by using just a few fuse-programmable controllers (FPCs) and programmable-logic devices.*

Reprinted from EDN, October 2, 1987
© 1987 CAHNERS PUBLISHING COMPANY
a Division of Reed Publishing USA.

You can implement VME Bus control in your computer system by embedding the bus-control functions in state machines that comply with the VME Bus protocol.

your first step is to consider the VME Bus protocols. These protocols specify the steps your circuitry must take to perform any bus-related operation, such as the transfer of data between two modules. You can describe these protocols by means of flow diagrams that show how the various interface signals reflect the interactions between the communicating modules. **Fig 2,** for example, shows how the priority bus arbiter resolves simultaneous bus requests from two modules that use the same request line.

Next, you analyze the flow diagrams to pick out the functions that can be incorporated into FPCs or PLDs, and you define state machines for those functions. You can describe the state machines abstractly, by Boolean logic equations, or diagrammatically, by means of standard flow-chart symbols (eg, rectangles to represent processes and diamonds to represent control-flow decisions). Finally, you must write programs for the state machines in a high-level or assembly language. You repeat this process to design each of the four main types of VME Bus controllers: bus arbiters, masters, slaves, and interrupt handlers.

Before any master module can perform a data transfer, it must request control of the data bus from the bus arbiter; the arbiter must check to see whether the bus is free and must resolve any contention between two masters of equal priority. For example, consider a priority-option bus arbiter implemented on a single PLD. The arbiter will monitor the four bus-request lines ($\overline{BR}_{0\text{-}3}$) and assign the highest priority to $\overline{BR}_3$.

As the flow diagram **(Fig 2)** shows, the arbiter grants the bus to the requesting module that's using the highest active request line, which is $\overline{BR}_1$ in this example. To enable your bus arbiter to resolve simultaneous bus requests from two or more modules that use the same request line, you must daisy-chain the associated bus-grant signal to all the devices using that request line. Therefore, the arbiter must be in the first slot of the VME Bus system. The module that's closest to the bus arbiter will have the highest priority: If it requests the bus, it will receive the bus grant and lock out any modules farther down the chain **(Listing 1).** The AND/OR array of the PLD processes all bus requests in parallel, so that priority arbitration is complete in only one clock cycle.

Priority arbitration options

Sometimes, the arbiter must force the current bus master to relinquish control of the bus; this situation occurs, for example, when a bus master of higher priority initiates a bus request. The arbiter examines the priorities of both the current bus master and the new requester. If these conditions meet the predefined bus-clear conditions, the arbiter asserts the bus-clear signal ($\overline{BCLR}$).

LISTING 1

```
IF (/BBSY*(BR0+BR1+BR2+BR3)) THEN  "if bus not busy and a       "
    BEGIN                          "  request line is active    "
    IF (BR3) THEN                  "if BR3 is active, grant     "
        BG3IN := 1 ;               "  bus to device on BR3      "
    IF (/BR3*BR2) THEN             "activate bus grant          "
        BG2IN := 1;                "  daisy chain 2             "
    IF (/BR3*/BR2*BR1) THEN        "if BR1 is active and BR3"
        BG1IN := 1;                "  and BR2 are not, then     "
                                   "  grant bus to device       "
                                   "  using request line 1      "
    IF (/BR3*/BR2*/BR1*BR0) THEN   "BR0-3 and BG0IN to
        BG0IN := 1 ;               "  BG3IN are active low      "
    END ;

    IF (/BBSY*BG1IN) THEN "if BG1IN is asserted in response to a request "
        BG1IN := 1;           "over line BR1, then continue asserting BG1IN"
                              "until requesting device drives BBSY active   "
```

LISTING 2

```
IF (BBSY*(BR0+BR1+BR2+BR3)) THEN
    BEGIN
    IF (BR3*(/(MASTER[1:0] = 3) ) ) THEN
        BCLR := 1;    "assert BCLR if MASTER < > 3"
    IF (/BR3*BR2*(/(MASTER[1:0] = 3) ) ) THEN
        BCLR := 1;
    IF (MASTER[1:0] = 0) THEN
        BCLR := 1;
    END;
```

In the design in **Fig 1,** the bus arbiter keeps track of the current bus master's priority by recording the bus-request line that was used to gain control of the bus. For example, if the current bus master used $\overline{BR}_2$ to gain control of the bus, it sets two output registers called Bus_Master to the binary value 2. Either of the two following conditions will activate $\overline{BCLR}$:

- The value in Bus_Master is 2, 1, or 0 and the active bus request line is 3 or 2.
- The value in Bus_Master is 0, and any bus-request line is active.

When the value in Bus_Master is 3, the arbiter will not honor any bus requests until the Bus Busy line ($\overline{BBSY}$) is high. **Listing 2** gives the logical expression of these conditions. Once having asserted $\overline{BCLR}$, the arbiter holds this line in the active state until the current bus master releases $\overline{BBSY}$.

Be sure to define the $\overline{BCLR}$ in such a way that uninterruptible devices can use bus-request line 3, which has the highest priority. Devices that temporari-

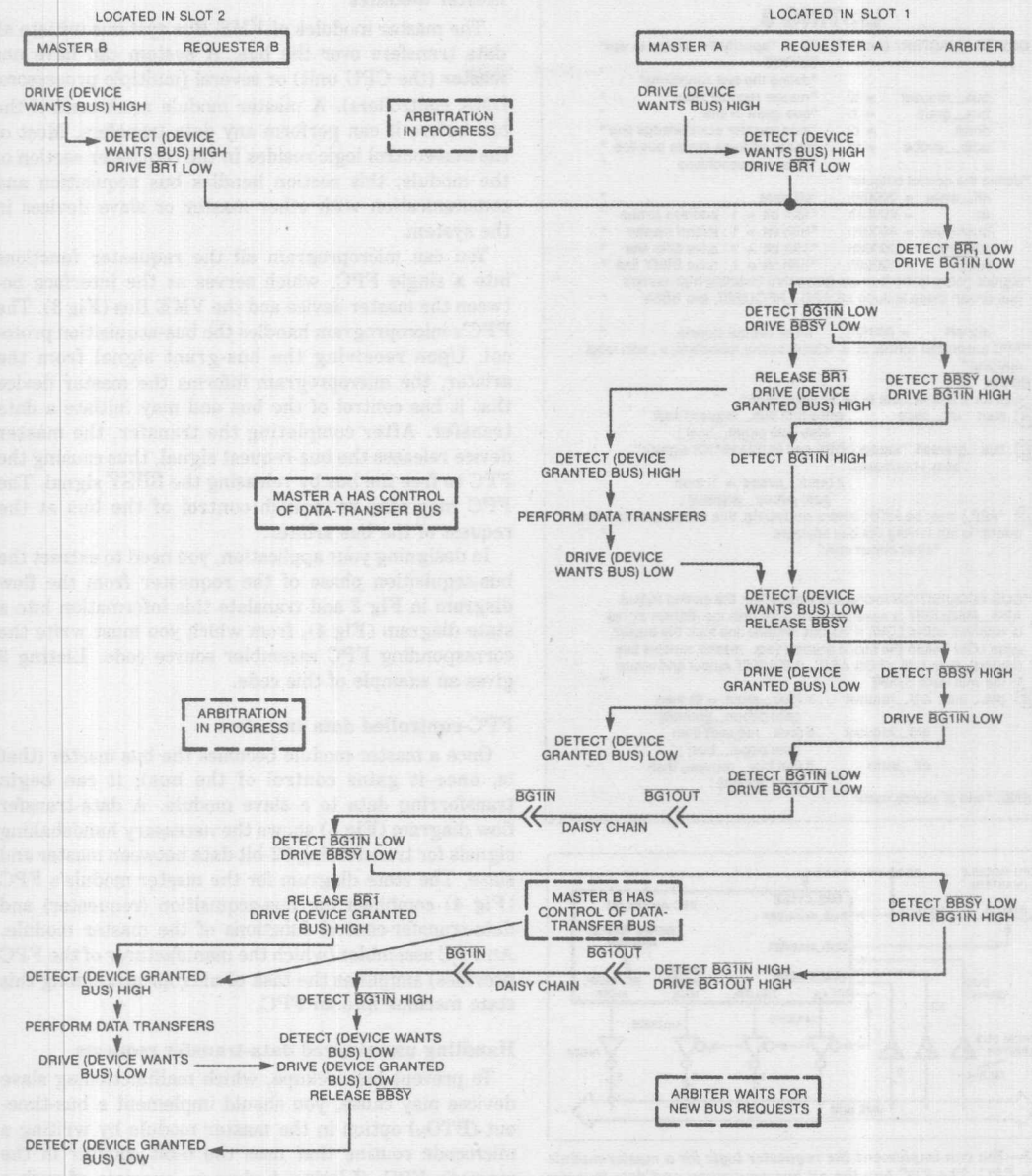


Fig 2—Flow diagrams help you define the VME Bus protocols. *This flow diagram illustrates how the bus arbiter chooses between requesters that have the same priority level.*

LISTING 3

```
DEVICE "MASTER" (Am29PL141)     "specify the device to use"
                                DEFINE
                                "define the test conditions"
    bus_request     = t0        "master request input          "
    bus_grant       = t1        "bus grant in line             "
    dtack           = t2        "data transfer acknowledge line"
    addr_strobe     = t3        "check address strobe bus line "
    : : : : : : : : :           "other test conditions         "
"define the control outputs"
    off_state  = 0000#h         "off state                     "
    as         = 8000#h         "16th bit = 1 : address strobe "
    busmaster  = 4000#h         "15th bit = 1 : inform master  "
    arb_req    = 2000#h         "14th bit = 1 : drive BRn line "
    bbsy       = 1000#h         "13th bit = 1 : drive BBSY line"
"signals going to the bus are gated thru inverting high-current
 bus driver; these include AS,ARB_REQUEST, and BBSY          "
    :
    signal1    = 0001#h ;       "other control signals         "
"FPC assembler format is = <label:output,statement;>, with label
 optional"
BEGIN
"wait for a bus request to be asserted high"
(A) start : off_state           , while (not bus_request) wait
                                  else goto pl(get_bus) ;
(C) bus_granted  "assert BBSY and BUSMASTER signals"
        : bbsy+busmaster
                                , if (addr_strobe = 1) then
                                  goto pl(bus_granted) ;
(D) "AS(L) must be HIGH before continuing; this means previous bus
 master is not driving the bus anymore                       "
        :    "other statements"
        :
        :
"BUS ACQUISITION microcode subroutine: the control output
 ARB_REQUEST is asserted (BRn line) until the /BGINn signal
 is received active LOW; if the bus request line from the master
 goes LOW before the bus is granted (e.g., master cancels bus
 request), then turn off the ARB_REQUEST output and return
 to the wait state START                                     "
(B) get_bus: arb_request        , if (bus_grant = 0) then
                                  goto pl(bus_granted) ;
             arb_request        , if (bus_request) then
                                  goto pl(get_bus) ;
             off_state          , if (not bus_request) then
                                  goto pl(start) ;
END. "end of source code"
```

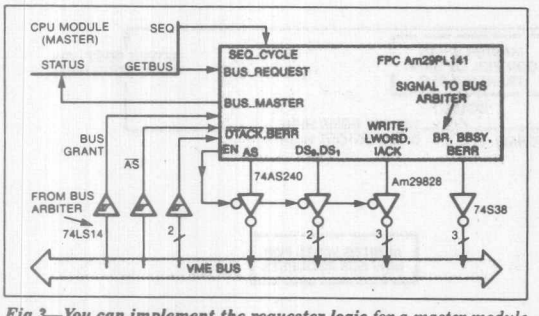


*Fig 3—**You can implement the requester logic** for a master module in an FPC. The FPC handles all bus-acquisition and data-transfer protocols.*

ly can be suspended to accommodate interrupts and higher-priority operations should be assigned to bus-request line 0. The BCLR conditions will vary with your application, but you can modify them just by redefining the high-level logic expressions.

Master modules

The master modules of VME Bus systems initiate all data transfers over the bus. A system can have one master (the CPU unit) or several (multiple processors, DMA controllers). A master module must control the bus before it can perform any data transfers. Most of the bus-control logic resides in the requester section of the module; this section handles bus acquisition and communication with other master or slave devices in the system.

You can microprogram all the requester functions into a single FPC, which serves as the interface between the master device and the VME Bus (**Fig 3**). The FPC's microprogram handles the bus-acquisition protocol. Upon receiving the bus-grant signal from the arbiter, the microprogram informs the master device that it has control of the bus and may initiate a data transfer. After completing the transfer, the master device releases the bus-request signal, thus causing the FPC to free the bus by releasing the $\overline{\text{BBSY}}$ signal. The FPC may also relinquish control of the bus at the request of the bus arbiter.

In designing your application, you need to extract the bus-acquisition phase of the requester from the flow diagram in **Fig 2** and translate this information into a state diagram (**Fig 4**), from which you must write the corresponding FPC assembler source code. **Listing 3** gives an example of this code.

FPC-controlled data transfers

Once a master module becomes the bus master (that is, once it gains control of the bus), it can begin transferring data to a slave module. A data-transfer flow diagram (**Fig 5**) shows the necessary handshaking signals for transferring 32-bit data between master and slave. The state diagram for the master module's FPC (**Fig 4**) combines the bus-acquisition (requester) and data-transfer-control functions of the master module. An FPC assembler (which the manufacturer of the FPC provides) simplifies the task of microprogramming this state machine into an FPC.

Handling unanswered data-transfer requests

To prevent bus lockups, which malfunctioning slave devices may cause, you should implement a bus-time-out (BTO_N) option in the master module by writing a microcode routine that uses the 6-bit counter in the master's FPC. (**Listing 4** gives an example of such a routine.) The bus-time-out option permits a bus master to abort a data-transfer cycle if the slave does not respond within n microseconds.

At the start of every data-transfer operation, the FPC microprogram loads a value into the 6-bit counter, tests for the data-transfer acknowledge signal ($\overline{\text{DTACK}}$), decrements the counter and tests it for zero,

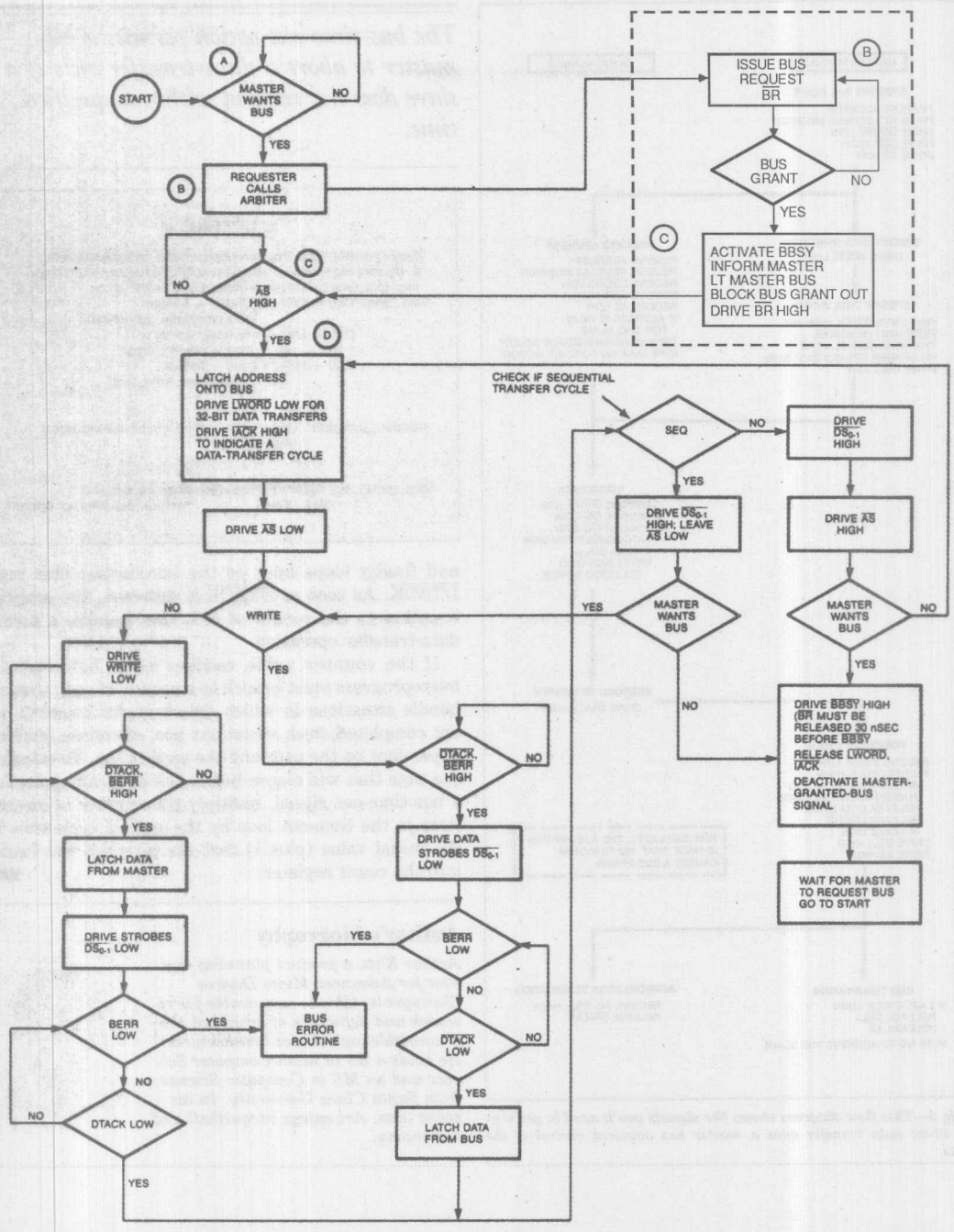


***Fig 4—You can derive detailed state-machine diagrams** from your flow diagrams. In this diagram for a master requester, the bus-acquisition phase is surrounded by a dashed line. Note the close correspondence between the state diagram and the assembly-language program in **Listing 3**.*

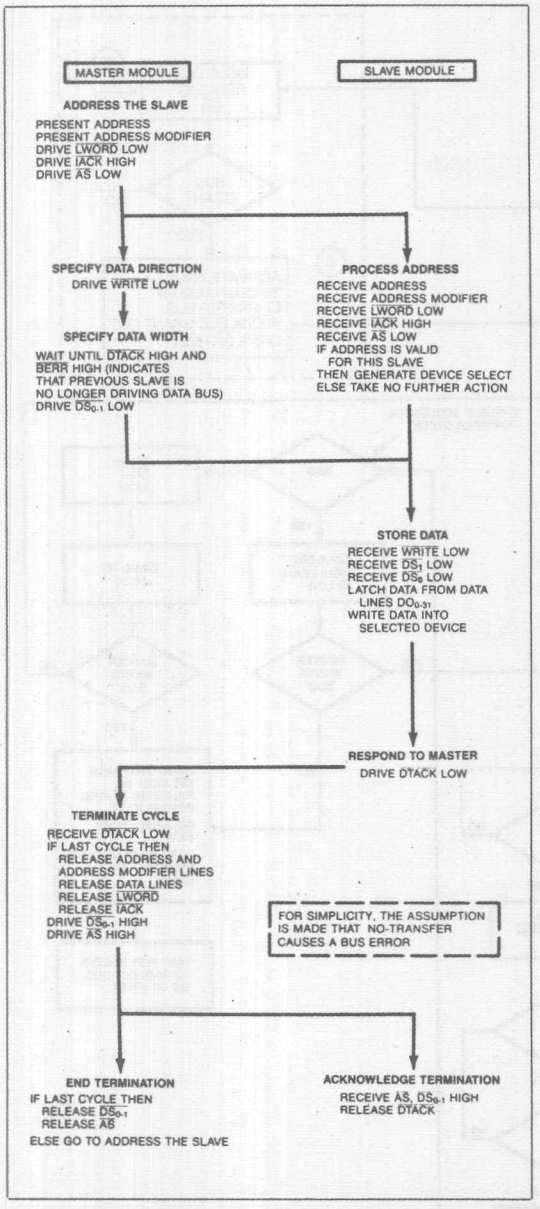


***Fig 5—This flow diagram shows the signals you'll need** to perform a 32-bit data transfer once a master has acquired control of the bus.*

The bus-time-out option permits a bus master to abort a data-transfer cycle if a slave does not respond within a specified time.

LISTING 4

```
"Begin transfer : Bus time out loop contains instructions 1 and
 2. If count reg is loaded with 63 and FPC is running at 20 MHz,
 then total time before bus times out is (63+1)*2*50 ns          "
wait_loop: DS0 + DS1 , if (DTACK = 0) then                       "1"
                    goto pl(transfer_complete) ;
           DS0 + DS1 , while (creg < > 0)                        "2"
                    loop to pl(wait_loop) ;
           DS0 + DS1 , if (creg = 0) then
                    goto pl(bus_time_out)
                    .
                    .
                    .
transfer_complete: "goto to this section if handshaking signal
                    was received"
                    .
                    .
                    .
bus_time_out: "execute this section when counter is 0"
              BTO_signal, . . . . . "activate bus time out signal"
```

and finally loops back to the instruction that tests $\overline{\text{DTACK}}$. As soon as $\overline{\text{DTACK}}$ is detected, the program branches to the section of code that handles a normal data-transfer operation.

If the counter value reaches zero, however, the microprogram must branch to a section of code that can handle situations in which data-transfer requests are not completed; such situations are, of course, entirely dependent on the user and the application. To calculate the time that will elapse before the program generates a bus-time-out signal, multiply the number of instructions in the time-out loop by the system cycle time by the initial value (plus 1) that the program has loaded into the count register. **EDN**

Author's biography

Arthur Khu, a product planning engineer for Advanced Micro Devices (Sunnyvale, CA), is responsible for research and definition of advanced programmable-logic-device architectures. He holds a BS in Math/Computer Science and an MS in Computer Science from Santa Clara University. In his spare time, Art enjoys racquetball and astronomy.

FPCs and PLDs implement VME Bus slave controllers

Designer's Guide to VME Bus Control—Part 2

By using fuse-programmable controllers (FPCs) and PLDs, you can implement VME Bus control in your computer system with a minimum of hardware. Part 1 (October 2, pg 187) of this 2-part series described the bus-arbitration process and control functions and showed how to implement the VME Bus data-transfer protocol in an FPC. This second and final part describes the implementation of similar controllers for slave modules and interrupt handlers, and it provides tools for programming the FPC.

Arthur Khu, *Advanced Micro Devices*

When you offload a system's bus-control functions from the CPU to a hardware bus controller, you considerably reduce the time these functions require, thereby improving the data-transfer rate over the bus. You can substantially reduce the cost and chip count of such a bus controller by implementing the bus-control logic in VLSI devices such as fuse-programmable controllers (FPCs) and programmable logic devices (PLDs).

The VME Bus protocol requires that all data transfers over the bus be initiated by a master module. Your system can include several master modules, such as CPUs or DMA controllers; a bus-arbiter module arbitrates simultaneous data-transfer requests from two or more masters. No data transfer can take place until the requesting master has been given control of the bus by the bus arbiter.

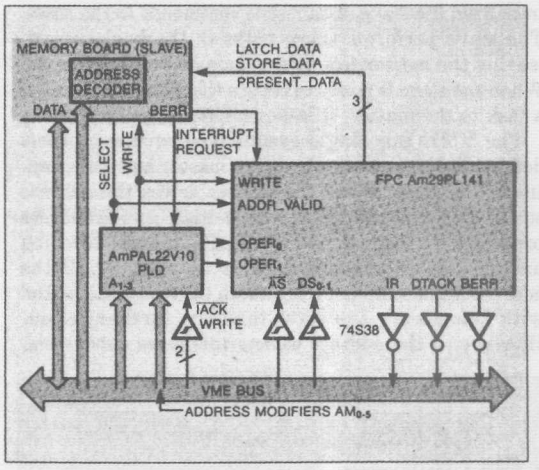


Fig 1—This device controller, *which is implemented with an FPC and a PLD, handles data-transfer and interrupt protocols for a slave device in a VME Bus system.*

Because a slave can't initiate a data transfer, your system must include an interrupt mechanism that allows a slave to request service from a master. You can implement such a mechanism in your system by designing a slave subsystem like the one in **Fig 1.** This slave subsystem comprises a slave module and a slave interrupt controller. The slave interrupt controller, which consists of an Am29PL141 FPC and an AmPAL22V10 PLD, serves as the interface between the slave subsys-

2

The interrupt controller for a slave device implements the interrupt protocol in hardware.

tem and the VME Bus structure. Once the master has initialized the slave, it can issue commands to the slave. The slave performs these tasks in the background, leaving the master free to continue its own operations. When the slave is ready to return the status or result of a task to the master, it issues an interrupt request.

The VME Bus single-cycle data-transfer protocol defines the interactions between master and slave controllers (**Fig 2**). When the master drives the address strobe ($\overline{AS}$) low, the slave controller latches the address presented on the bus, and a separate address-decoding unit on the slave board decodes the address. If the address is not valid (ie, if it's not in the range associated with this slave), the slave takes no further action. However, if the address selects this slave subsystem,

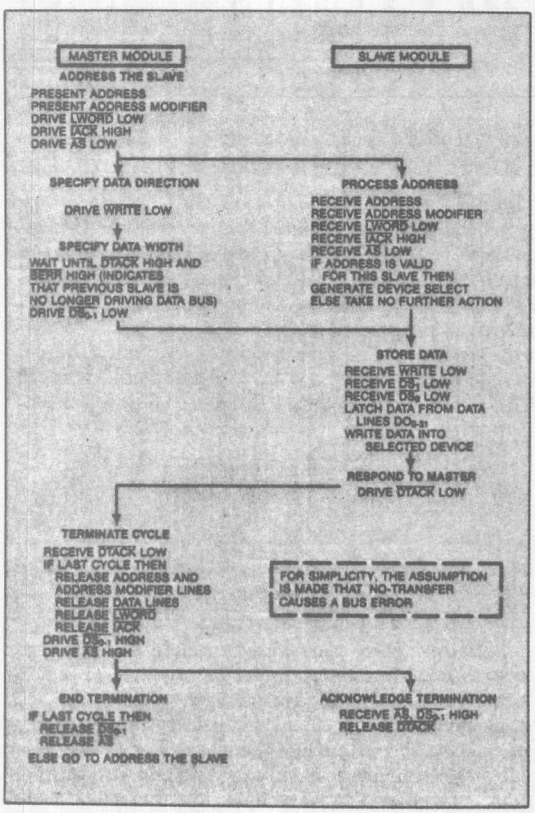
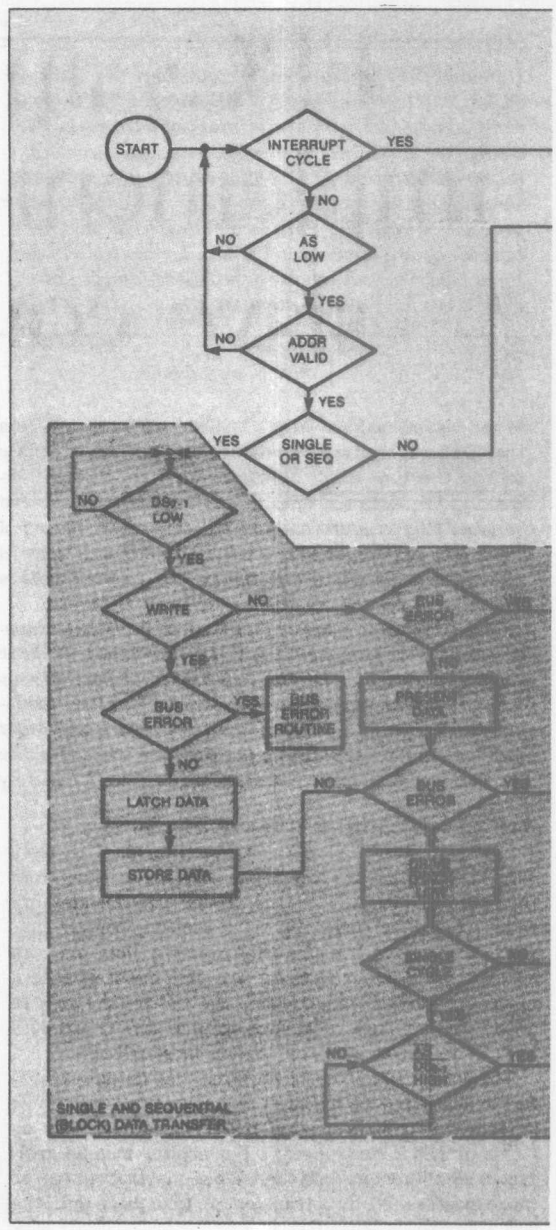


***Fig 2—The slave controller recognizes its own address** and receives a data word from the bus master in this flow diagram for single-cycle transfer.*



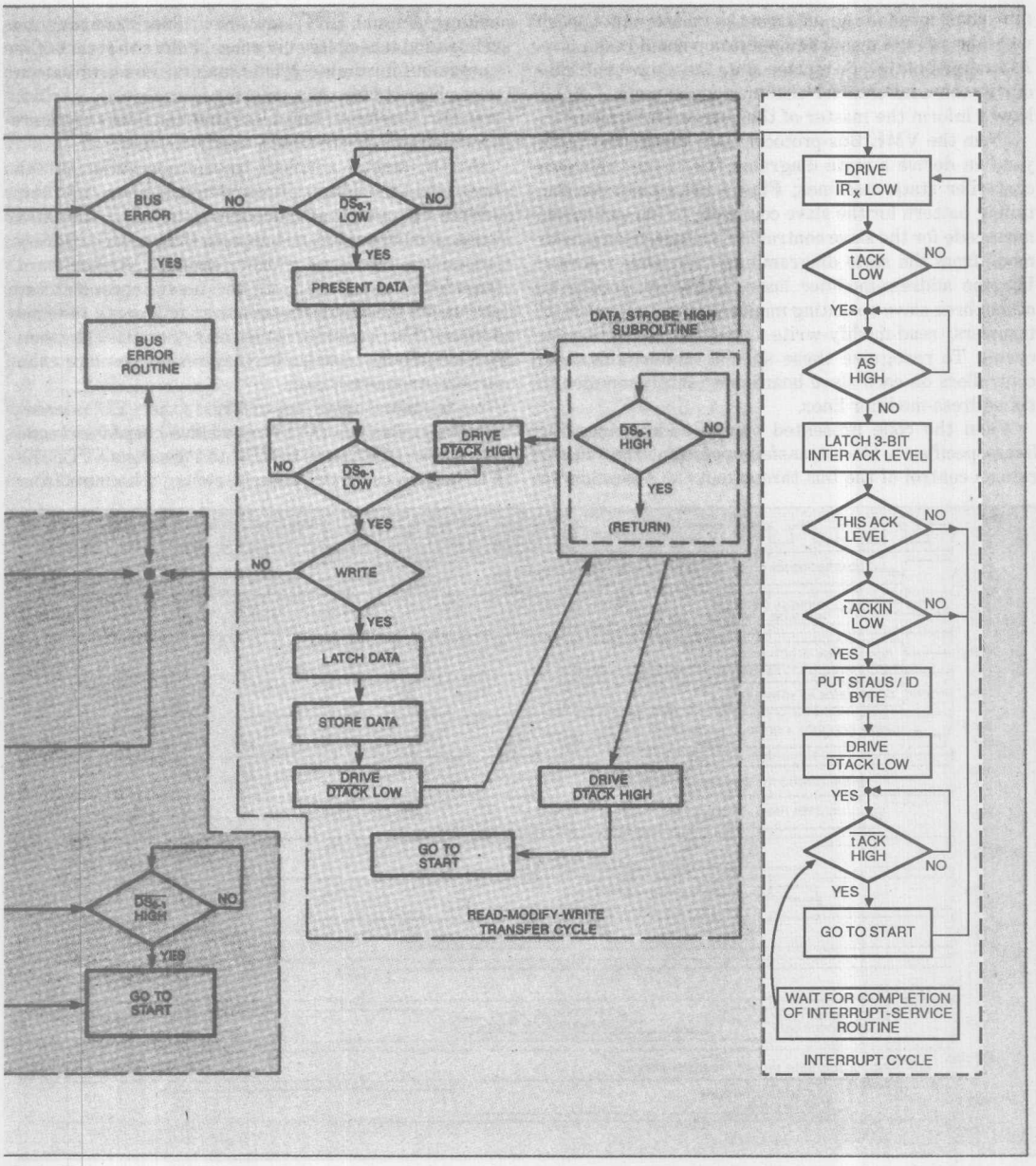


*Fig 3—**This state-machine diagram** for **Fig 1's** slave controller is derived from the flow diagram in **Fig 2.** The state machine has four modes, which handle interrupts as well as single, block, and read-modify-write data transfers.*

data transfers over the bus be initiated by a master module.

the slave looks for signals from the master that specify whether a read or a write operation should take place. After presenting or storing data, the slave controller drives the data-transfer-acknowledge signal ($\overline{DTACK}$) low to inform the master of the successful transfer.

From the VME Bus-protocol flow diagrams (**Fig 2**), you can derive a state diagram (**Fig 3**) for the slave-controller state machines; **Fig 4** shows the resulting timing pattern for the slave controller. You can develop microcode for the slave controller's single-cycle transfer mode from the state diagram and the timing pattern. Use the address-modifier lines ($AM_{0\text{-}5}$) to specify the other three slave operating modes (sequential, or block, transfers; read-modify-write transfers; and interrupt cycle). To recognize these special modes, the slave controllers on each slave board constantly monitor the six address-modifier lines.

When the code presented on the address-modifier lines specifies a block-transfer operation, the master retains control of the bus throughout the operation by holding $\overline{AS}$ and $\overline{BBSY}$ low. For a block transfer, bus arbitration takes place only once, before the start of the operation; for single-cycle transfers, bus arbitration takes place before the transfer of each word. A block transfer, therefore, takes less time than does the corresponding number of single-word transfers.

At the start of a block-transfer operation, all the slaves load the address presented on the bus into their address counters and decoders, but only the slave whose memory range encompasses the decoded address responds to the data-transfer request. As each word transfer is completed, all the slaves increment (or decrement) their address counters and decode the new address. This procedure is necessary because the memory block being transferred may reside on more than one slave memory board.

In the slave subsystem in **Fig 1**, the PLD decodes control signals from the bus and slave board and sends two signals, $OPER_0$ and $OPER_1$, to the slave's FPC. The FPC decodes these two signals, along with inputs from

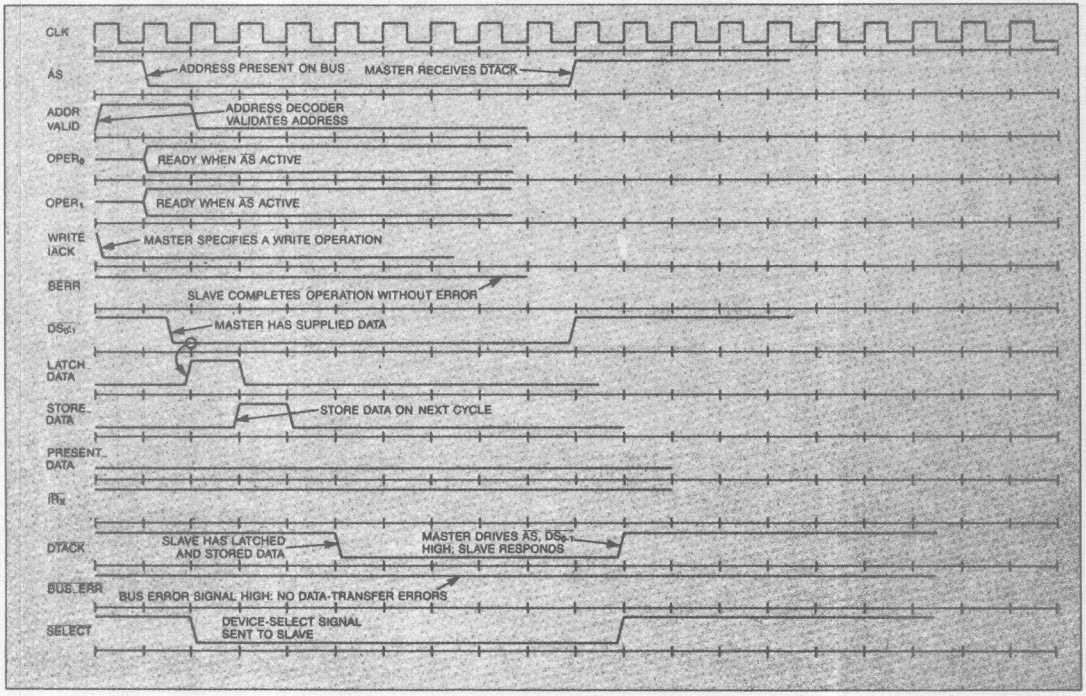


Fig 4—You'll need to generate a timing diagram *for each of the slave controller's operating modes. This diagram shows the timing for a single-cycle data transfer.*

the slave, bus, and address-decoding units on the slave board, to determine which of the four possible slave operating modes to execute. The four modes, which are designated by binary codes, specify the following operations:

- (00) Perform a single-cycle transfer
- (01) Perform a sequential-cycle transfer
- (10) Perform a read-modify-write cycle
- (11) Perform an interrupt cycle.

If $OPER_0$ and $OPER_1$ are both high, the FPC operates as an interrupter by branching to an interrupt subroutine (**Fig 3**).

When the slave requests an interrupt, the FPC generates an interrupt-request signal and waits for the interrupt-acknowledge signal ($\overline{IACK}$) and daisy-chain signal ($\overline{IACKIN}$). When the data strobes become active, the PLD reads a 3-bit value from the address bus (A_{1-3}); this value indicates which interrupt-request line was acknowledged. The PLD decodes these three bits to determine whether their value matches its own request level; if it finds no match, no further action occurs. If it does find a match, however, the PLD routes a valid signal to the FPC, indicating that the interrupt handler has acknowledged the slave's interrupt request. The FPC then signals the slave board to put its status or identification byte on the data bus for the interrupt handler to use as an interrupt vector. The slave FPC waits in a loop until the interrupt handler drives the $\overline{IACK}$ signal high to signify that interrupt service is complete.

Besides containing master, slave, and bus-arbiter modules, a VME Bus system usually has an interrupt-handler module that handles external I/O or special system events (time-out or overflow errors, for example). You can reduce the logic complexity of the interrupt-handler module in your system by offloading some of the initial interrupt-recognition tasks to an interrupt-handling preprocessor (IHP). You can use a PLD as the IHP, programming it to preprocess interrupt requests, obtain control of the bus, and handle hand-

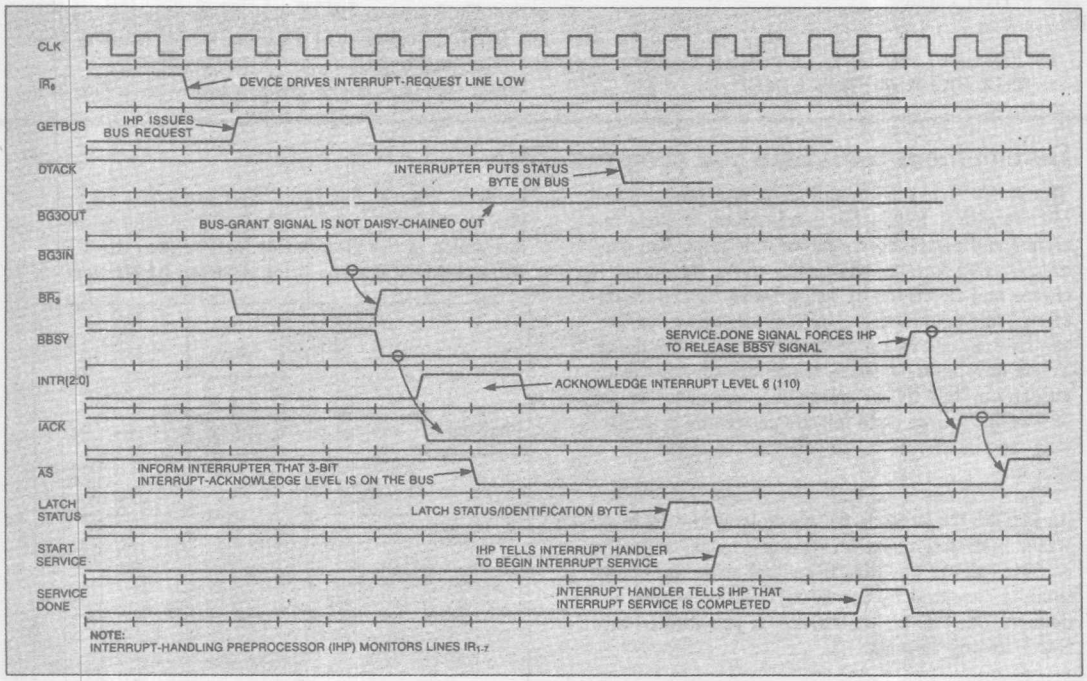


Fig 5—The interrupt-handling preprocessor *monitors the seven VME Bus interrupt-request lines ($\overline{IR_{1-7}}$), identifies the interrupting device, and tells the interrupt handler when to begin servicing the interrupt. This timing diagram shows the signal states that exist before and during the transfer of the identification and status bytes from the slave-interrupt subsystem to the interrupt handler.*

When the slave module is in block-transfer mode, bus arbitration takes place only once before the start of the operation.

shaking signals (such as interrupt-acknowledge signals). Only when the IHP latches the interrupt vector will control pass to the interrupt-handler module.

To define interrupt-request processing, bus acquisition, and the interrupt vector's transfer phase, you'll have to use logic equations written in high-level Boolean notation. In the design in **Fig 1**, the PLD monitors seven interrupt-request lines and four data-transfer control inputs, and it sends 10 control signals to the interrupt handler and the VME Bus drivers. The PLD monitors all interrupt-request lines according to the following logic equation:

```
IF (IR1 + IR2 + IR3 + IR4 + IR5 +                (1)
  IR6 + IR7) THEN
BR3 : = 1 ;
```

"THIS INTERRUPT HANDLER USES"
"THE BR3 REQUEST LINE"

If any interrupt-request line is active, the PLD asserts $\overline{BR3}$ to initiate the bus-acquisition phase.

The next step is to wait for the bus-grant-in signal. Only when $\overline{BGIN3}$ is active will $\overline{BBSY}$ be active. The logical expression of this condition is given in the following equations:

```
IF (BR3*/BG3IN) THEN                             (2)
        BR3 : = 1 ;

IF (BR3*BG3IN + BBSY*/SERVICE_DONE) THEN         (3)
        BBSY : = 1 ;
```

Eq 2 continually asserts $\overline{BR3}$ as long as $\overline{BG3IN}$ is not active; **Eq 3** asserts $\overline{BBSY}$ only when request line $\overline{BR3}$ and $\overline{BG3IN}$ are active, or if service is not complete after the IHP asserts $\overline{BBSY}$.

When the IHP is the bus master, it puts the 3-bit

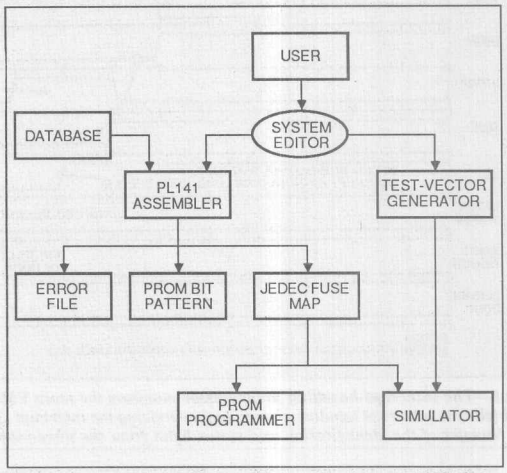
Development tools help you program FPCs

The Am29PL141 is a single-chip fuse-programmable controller (FPC) that can implement state machines and distributed control functions. You can express and functionally verify these state machines and functions by using three FPC-development tools provided by the manufacturer: an assembler, a test-vector generator, and a simulator. These development tools, which are written in C, run on an IBM PC or compatible computer under MS-DOS. Source code for the programs is available, so you can port the software to other systems.

The FPC has a control store that's resident in its PROM; the store is 64 words long and 32 bits wide. Instructions such as jumps, loops, and subroutine calls in the PROM control store are conditionally executed by the 20-MHz internal sequencer. Each 32-bit instruction is partitioned into the following format:

```
BITS:   1     5     1     3     6        16
FIELD: OE OPCODE POL TEST DATA OUTPUT
```

The output field contains the 16 control outputs that the FPC generates when it executes an instruction. The upper eight output bits are controlled by the output enable (OE) bit; they may be high, low, or disabled. The lower eight output bits are always enabled. The op-code field specifies which of the 29 possible instructions the FPC will execute if the condition selected by the test



***To program the Am29PL141 FPC to implement VME Bus-control functions,** you can use the assembler, test-vector generator, and simulator provided by the FPC's manufacturer. These development tools, which are written in C, run on an IBM PC or compatible computer under MS-DOS.*

interrupt-line-acknowledge value on the bus and asserts the data strobes. Upon receipt of the $\overline{\text{DTACK}}$ signal from the interrupter, the IHP strobes the status byte from the bus into a register on the interrupt-handler card. The IHP then informs the interrupt handler that a status/identification byte is ready and that the handler should begin servicing the interrupt. The IHP resolves interrupt priorities within a single clock cycle because all interrupt/input signal lines are processed in parallel by the logic array in the PLD. **Listing 1** shows how you'd express the procedure in a logic-description language.

As you can see from **Listing 1**, if both the $\overline{\text{IR7}}$ and the $\overline{\text{BBSY}}$ signals are active in section 1, then the 3-bit value generated by the IHP will be binary 111 (decimal 7), regardless of the state of the other interrupt lines. In section 2, if $\overline{\text{IR1}}$ and $\overline{\text{BBSY}}$ are active and the other six control signals are inactive, then the 3-bit output value will be binary 001 (decimal 1). In section 2, $\overline{\text{IR1}}$ is the highest priority line that is active.

The remaining interrupt-preprocessing steps complete the transfer of the interrupt-vector byte; this transfer is described logically in the following

LISTING 1

```
IF (BBSY) THEN
   BEGIN
(1) IF (IR7) THEN                "IR7 was active                "
       INTR[2:0] : = 7 ;         "3-bit value acknowledging
                                  interrupt line 7              "
    IF (/IR7*IR6) THEN           "IR6 active, IR7 inactive      "
       INTR[2:] : = 6 ;          "acknowledge interrupt line 6"
    IF (/IR7*/IR6*IR5) THEN      "IR5 highest priority          "
       INTR[2:0] : = 5 ;         "interrupt line active         "
                 :
                 :
                 :
(2) IF (/IR7*/IR6*/IR5*/IR4*/IR3*/IR2*IR1) THEN
       INTR[2:0] : = 1 ;         "acknowledge interrupt line 1"
END;
```

```
IF  (BBSY) THEN                                                    (4)
    BEGIN
    IACK := 1;                   "begin interrupt acknowledge daisy"
(X) IF (DTACK) THEN              "chain; if device sends data      "
       LATCH_STATUS := 1;.       "transfer acknowledge (DTACK)     "
                                 "signal, then latch the status    "
    END:
IF  (IACK) THEN
    AS := 1;          "assert the address strobe signal to inform the
                       interrupter that the interrupt acknowledge
                       level is ready"
```

field is true. The polarity bit (POL) determines whether a high input or a low input represents true for the test input selected. You can use the data field as an argument for the op-code field. For example, if you want the FPC to jump (op code 25) to location 34 (22_{HEX}) when the second bit in the test field is high (true), you put the following 32-bit word into the control store (though you'll also have to define the OE bit and the output bits):

```
OE 11001 0 010 100010 OUTPUT
```

The assembler simplifies system design by allowing you to use a high-level language. Instead of specifying the bit patterns shown above, you can write the following section of code:

```
OUTPUT , IF (T2 = 1) THEN
                GOTO PL (34);
```

The assembler translates this statement into the appropriate bit patterns; when the FPC executes the instruction, it generates the bit pattern defined by the symbol OUTPUT.

Every microinstruction written in assembly language follows this format:

```
< LABEL : >  OUTPUT , STATEMENT ;
```

The label field, which is optional, simplifies the writing of subroutine calls and conditional branch instructions, which you can express in IF-THEN-ELSE, WHILE-DO, or COMPARE forms. Once you've written all the instructions with the editor and translated them to executable form with the assembler, you can cause the assembler to generate a JEDEC fuse map.

Before physically programming the PROM control store that you've designed, you can test your design with the help of the test-vector generator and simulator. The test-vector generator produces test vectors, from a user-generated truth table, and converts them into a JEDEC-standard test-vector file that the simulator can use.

The simulator performs two important functions: It verifies the control flow of your design, and it checks the fuse map that the assembler generates. By using the simulator's interactive mode, you can inspect, modify, or preset any of the internal registers in the FPC. The simulator also has single-stepping and break-point features so that you can trace the operation of your design in detail.

An interrupt-handling processor can offload some of the initial interrupt-recognition tasks of your system's interrupt-handler module.

Once the assertion of the $\overline{\text{DTACK}}$ signal indicates the successful transfer of the 8-bit status or identification byte, the IHP instructs the interrupt-handler module to begin the interrupt-service routine. This instruction from the IHP is logically defined as follows:

```
IF (BBSY*LATCH_STATUS + BBSY*                    (5)
   START_SERVICE*/SERVICE_DONE) THEN
START_SERVICE : = 1;
```

This definition states that the IHP constantly asserts the START_SERVICE signal until the interrupt handler generates a SERVICE_DONE signal, at which point the IHP drives START_SERVICE low. The logic described above generates the timing diagram shown in **Fig 5.**

Development tools provided by FPC and PLD manufacturers simplify the task of programming these devices as VME Bus controllers. Once you've analyzed the bus protocols and converted these into state-machine diagrams, you can write assembly-language programs and high-level logic equations to describe the state machines. The assembler and logic software will then process these programs and equations to fit into the FPC or PLD. For a summary of the programming tools available for the Am29PL141 FPC, see **box,** "Development tools help you program FPCs." **EDN**

Author's biography

Arthur Khu, a product planning engineer for Advanced Micro Devices (Sunnyvale, CA), is responsible for research and definition of advanced programmable-logic-device architectures. He holds a BS in Math/Computer Science and an MS in Computer Science from Santa Clara University. In his spare time, Art enjoys racquetball and astronomy.

Fuse-Programmable Chip Takes Command of Distributed Systems

The first fuse-programmable controller eliminates bulky and expensive designs, freeing distributed intelligence to carve out a greater niche for itself.

In much the same way that cars and highways spawned the suburbs, standard microprocessor buses and add-on boards have distributed processing intelligence, revolutionizing the design of digital systems. Breaking systems into independent modules shortens the design cycle, eases upgrades, and accelerates fault diagnosis. But despite these advantages, an essential element has been missing: a one-chip controller geared specifically to the needs of distributed intelligence.

Without that critical ingredient, engineers have been forced to turn to less than optimal solutions. One approach relies on boards packed with as many as 35 SSI and MSI devices. Another tack is to go with a powerful—yet costly—VLSI chip. Alternatively, a programmable logic device can be pressed into service, but such circuitry lacks the computing power to control peripherals.

The missing element, a fuse-programmable controller chip, is now here. By mixing intelligence and control, the Am29PL141 stakes out new territory for distributed systems. The 20-MHz IC combines for the first time all of the elements of an intelligent microcode controller. Its powerful sequencing logic steps through the controller's 64-by-32-bit pipelined PROM. That fuse-programmable memory stores a user-defined microprogram drawn from a set of 29 microinstructions, including a repertoire of jumps, multiple branches (or case statements), and subroutine calls. All can be executed conditionally, depending upon the outcome of one of eight tests. In addition, a serial shadow register on the 28-pin chip helps designers diagnose system troubles right down to a particular IC. (In the past, expediency often dictated that complex trouble-shooting be avoided for as long as possible.)

Om Agrawal and **Deepak Mithani**
Advanced Micro Devices Inc., 901 Thompson Pl., P.O. Box 3453, MS 47, Sunnyvale, CA 94088; (408) 749-2903.

Reprinted with permission from Electronic Design, Vol.33, No.24, Copyright Hayden Publishing Co., Inc., 1985.

Four basic blocks

The controller comprises four main functional blocks. Three of them—the microaddress control logic, condition code selector, and microinstruction decoder—form the cornerstone of the controller, the address sequencer. The fourth is a microprogram memory (64 by 32 bits) with a pipelined register and serial shadow register (Fig. 1).

For the most part, the elements of the address sequencer are fairly typical. Nevertheless, the way in which they are organized and connected, as well as the instruction set, make the chip unique. For example, the microaddress control portion of the sequencer contains one register for counting loops and another for stacking subroutine return addresses. Yet either register can be employed to double the capacity of the other. Consequently, the chip can nest two levels of loop counting or two levels of subroutine branching (the instruction set reflects those abilities). And the high degree of interaction between elements, particularly within the microaddress control logic, makes necessary a highly sophisticated micro-

Electronic Design • October 17, 1985

instruction set.

The microaddress control logic is the brain of the address sequencer, since it generates the addresses that access the microinstructions. At any time, the address that is called depends on the preceding instruction and the outcome of any conditional tests.

Within the control logic, a program-counter multiplexer supplies the PROM's 6-bit address. The multiplexer takes the address from a microprogram counter, incremented program counter, branch control logic, or subroutine register. Because the program counter contains the address of the currently executing instruction, that instruction is executed again when the program counter is selected as the address source. As a result, the counter plays a fundamental role in tallying loops and executing "wait until true" instructions.

The incrementer holds the next address in the sequence, and is the expected source when no jumps are executed and no branch or subroutine conditions exist. When conditional statements like "if . . . then . . . else" and multiple branches pass the required tests, or when unconditional jumps are executed, the branch control logic supplies the address. Finally, when the program calls a subroutine, the subroutine register supplies the necessary address.

A multiplexer selects one of three address sources. If only one stack level is needed, the value stored in the subroutine register is chosen. When the count register feeds the subroutine register, however, it furnishes an additional stack level. The third source, the incrementer, supplies the subroutine's return addresses.

Doing double duty

If not needed for a second subroutine level, the count register can, among other functions, execute iterative loops and time external events. To accomplish the former, the controller loads the register with the number of iterations to be run. Each iteration decrements the register until it reaches zero. The zero-detection logic associated with the counter informs the chip's microinstruction decoding logic when the register "bottoms out."

Using the same logic, an instruction can be repeated a set number of times. Repeated executions of the same instruction is a simple way to insert wait states and, therefore, build an interface to different microprocessors and peripherals.

The count register is loaded from any of four sources: a decrementer, for normal loops; an instruction field; the subroutine register; and the branch-control logic. The last derives a 6-bit value from a data field in a microinstruction.

The branch-control logic, a powerful block within the sequencer, calculates the 6-bit value either by applying the microinstruction data field directly or by using it to mask the chip's six test inputs, T_0 through T_5. In the second case, the masked input actually becomes the branch address. Moreover, either the data field or masked test value serves as both a branch address and a count value.

The same control logic also compares the masked test inputs to a constant in a microinstruction field. The outcome of this check affects a flip-flop. The latter's condition itself becomes a factor in deciding conditional branch and subroutine instructions. If a match occurs, the flip-flop is set. Alternatively, the flip-flop remains unchanged if there is no match. Because the flip-flop does not change when there is no match, it is particularly useful for comparing ASCII characters and other 6-bit fields, as well as for successively checking the chip's test inputs.

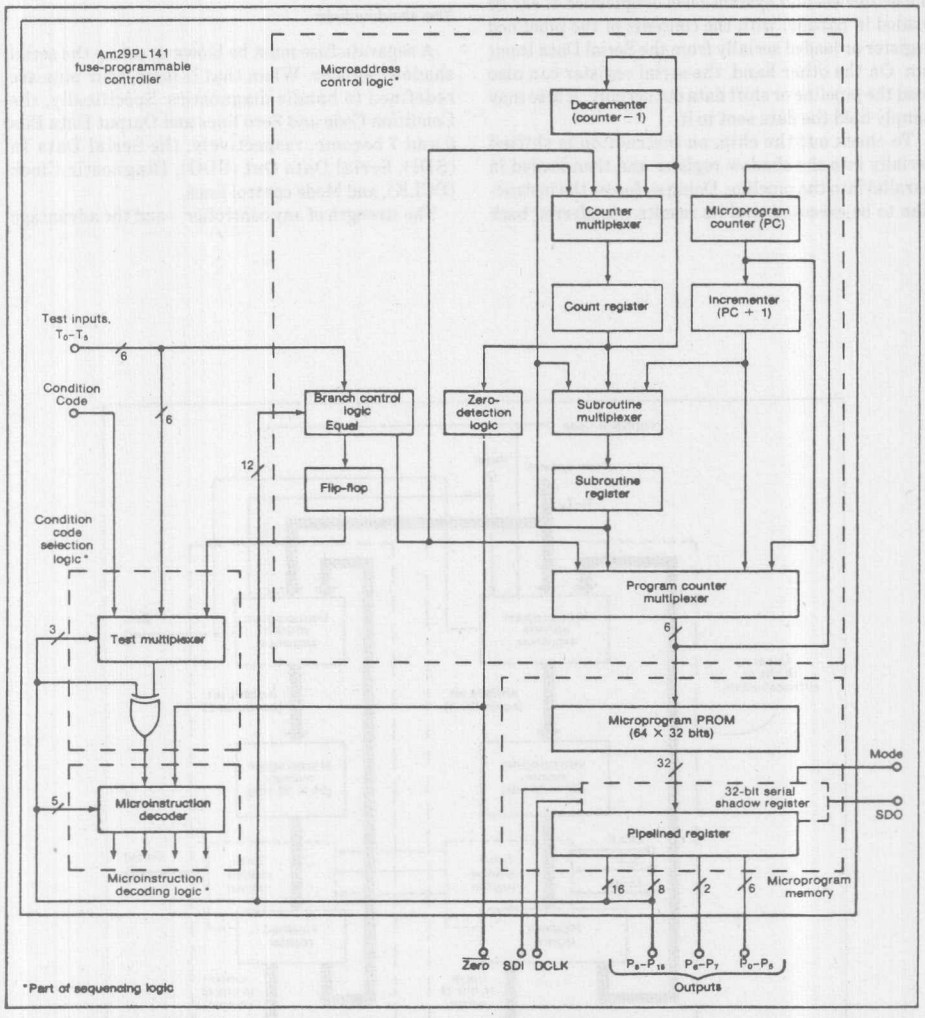
Controlling conditions

A set flip-flop is one of the eight aforementioned tests that fulfills a conditional branch or subroutine. Through its condition-code selection logic, the controller is able to check each of its six test input lines, as well as the Condition Code input. Further, an exclusive-OR gate within the selection logic switches the meaning, or interpretation, of a test result. In other words, with no external hardware, a test condition can be asserted either when a match occurs or when one does not.

The final component of the chip's sequencer section is the microinstruction decoder. That programmable logic array generates the IC's internal control signals based on the microinstruction being executed and the test results reported by the condition-code logic.

The IC's fourth functional block comprises the fuse-programmable microprogram memory, pipelined register, and serial shadow register. The pipeline register is 32 bits wide, and stores the microinstruction being run. The next address is calculated by the sequencer and its contents is fetched from the microprogram memory. The upper 16 bits of the pipeline's output remain within the chip to sequence addresses and control internal functions.

Only the 16 low-order bits link to the outside, as user-defined control lines. Of these, the upper byte is



1. The 20-MHz Am29PL141 is the first complete microprogrammable controller chip, making it an important building block for distributed processing systems. Its powerful sequencing logic steps the controller through its pipelined PROM. The fuse-programmable memory is 64 by 32 bits.

can be cascaded readily if more than 16 control bits are needed (Fig. 2).

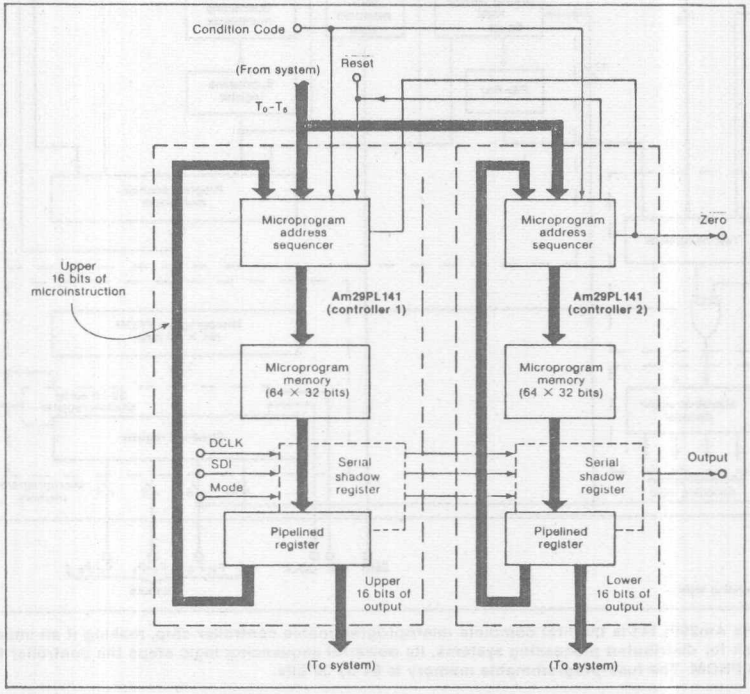
The serial shadow register, also 32-bits wide, simplifies device- and system-level diagnostics. It can be loaded in parallel with the contents of the pipelined register or loaded serially from the Serial Data Input pin. On the other hand, the serial register can also load the pipeline or shift data out serially. It also may simply hold the data sent to it.

To check out the chip, an instruction is shifted serially into the shadow register and then loaded in parallel into the pipeline. Doing so forces the instruction to be executed, and its results transferred back registers in a system are tied together, a series diagnostic loop is created that isolates a problem down to a single chip.

The shadow fuse

A separate fuse must be blown to set up the serial shadow register. When that is done, four pins are redefined to handle diagnostics. Specifically, the Condition Code and Zero lines and Output Data Bits 6 and 7 become, respectively, the Serial Data In (SDI), Serial Data Out (SDO), Diagnostic Clock (DCLK), and Mode control lines.

The strength of any controller—and the advantage



2. When an application calls for more than 16 control bits, two or more chips can be cascaded horizontally. The lower 16 bits of each of the chip's 32-bit microinstructions (of which there are 29) serve to control a system's components. Eight of these 16 control bits can be put into a high-impedance state under microinstruction control; the other 8 bits are always enabled.

of a microprogrammed system that employs it—lies in an engineer's ability to specify the sequence in which microinstructions are executed. To ensure that ability, the controller executes all the basic high-level constructs required for structured microprogramming. Its 29 op codes include sequential instructions, conditional instructions, dual branching forks, and multibranching case statements. Iterative executions, like For, While, When, and Until, round out the set. In addition, Jump, Jump to Subroutine, Loop, and Compare instructions allow designers to store very complex algorithms in the chip's 64-word memory.

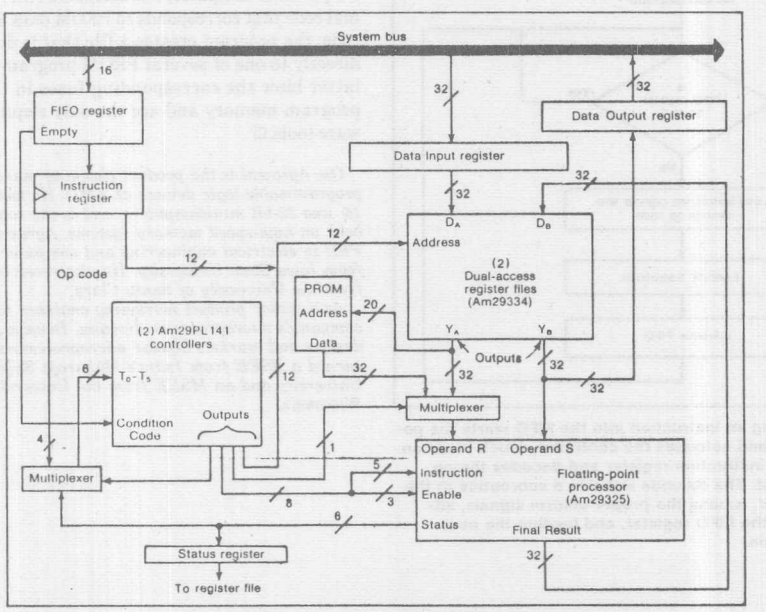
Instruction formats fall into two categories. The first is for general microinstructions; the second is for the chip's Compare instructions. The latter compare a 6-bit test input to a masked constant. The Compare instructions are well-suited for character searches, as well as key searches in a look-up table.

A single-precision, floating-point peripheral board (Fig. 3) presents a good example of the part that the controller plays in a distributed system. As a microprogrammed design, the peripheral serves as an add-on math accelerator card that plays with different hosts and buses. The controller orchestrates the actions of the floating-point processor, and various registers, register files, and memory chips.

Simple arithmetic

The processor is simple to use, partly because it incurs no pipeline delays. It conforms to IEEE and other industry standards, and takes only a single clock cycle to add, subtract, or multiply. It needs five cycles to divide, using the Newton-Raphson method that inverts one of the factors and multiplies. In operation, to divide X by Y the chip fetches the approximate inverse of Y from a PROM-based table and multiplies it by X. One or two iterations of this method increase the initial accuracy.

The floating-point board works with a microword

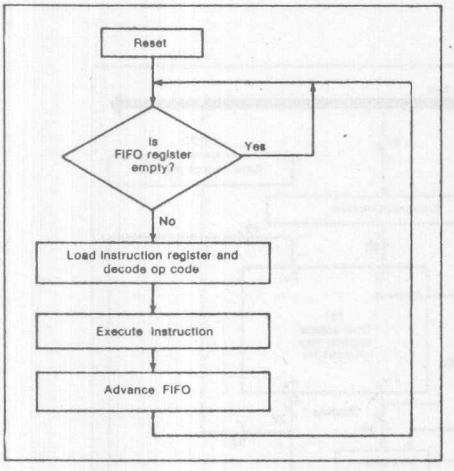


3. In a typical application, the controller oversees the workings of a floating-point processor board. Host instructions sent to the FIFO and instruction registers initiate subroutines in the controller that generate the signals that run the board. Two controllers are employed to supply the necessary number of control signals.

of at least 25 bits, 9 more than available with one controller. Thus the design employs two controller chips. The floating-point processor requires five command bits. Three are instructions and two select the input source. It also needs three control bits to enable its trio of data registers. Two other chips (each a dual-access four-port register of 64 words by 18 bits) temporarily store commands. They accept 12 register address bits (6 for source and 6 for destination) from the host or the controller's microprogram memory.

Data passes to and from the host through input and output registers on the board, which call for their own enable signals. Another bit is needed to advance the FIFO instruction register. One is necessary to enable and another to select a status word. (The floating-point chip supplies status information, which is available to the controller through its test inputs as well as to the host through the register file.) Seven control bits are left for miscellaneous tasks.

Operation begins when at least one 16-bit instruction is loaded from the host into the peripheral's FIFO register. The instruction consists of a 4-bit op code, a 6-bit source-register address, and a 6-bit address for the second source register, which also stores the results. Until an instruction is received, the controller is in the wait state (Fig. 4). When an instruction arrives, however, the FIFO's Empty signal activates the controller, which then reads the command from the FIFO into a separate instruction register. The controller also loads the instruction's op code into its test inputs. It then masks the two unused test bits and jumps to a subroutine that performs the operation specified by the op code. After completing it, the controller advances the FIFO register to load the next instruction.



4. Loading an instruction into the FIFO starts the peripheral and activates the controller, which loads an external instruction register and decodes the op code field. The op code initiates a subroutine in the controller, issuing the proper control signals, advancing the FIFO register, and loading the next instruction.

The peripheral executes up to 16 op codes. The first eight are single-cycle operations and identical to those of the floating-point processor. They consist of addition, subtraction, multiplication, and format conversion instructions. The remaining op codes are used to load, store, and divide data, and a multiple cycle instruction multiplies and accumulates values. The four remaining op codes can be defined by the user to implement application-related operations.

Software and hardware tools are a necessary part of such projects as the foregoing peripheral. The software assembles high-level microprograms and a JEDEC output file that specifies the fuse pattern to be burned into the PROM array. Currently, a program called Fuse Formatter, which runs on the IBM PC personal computer, lets designers enter hexadecimal code that corresponds to PROM data. From that code, the program creates a file that is downloaded directly to one of several PROM programmers. The latter blow the corresponding fuses in the microprogram memory and are the only required hardware tools. □

Om Agrawal is the product planning manager for programmable logic devices at AMD. He has designed 16- and 32-bit minicomputers, and is the coauthor of a book on high-speed memory systems. Agrawal holds a PhD in electrical engineering and computer science from Iowa State University. He also received an MBA from the University of Santa Clara.

As a senior product marketing engineer for the company's microprocessor division, Deepak Mithani, designs and markets bipolar microprocessors. He earned a BSEE from India's Maharaja Sayajirao University and an MSEE from the University of Wisconsin.

PAL Device Buries Registers, Brings State Machines to Life

Om Agrawal and Kapil Shankar
Advanced Micro Devices Inc., 901 Thompson Pl., P.O. Box 3453, Sunnyvale, CA 94088; (408) 732-2400.

Designers who build state machines from programmable logic devices (PLDs) must develop good juggling skills. The job usually takes a careful balancing of the I/O pins, the product terms, and the registers that store states and output bits. But because these resources all are limited in a PLD, performance and compactness often have to be sacrificed in favor of the desired function.

Ideally, state-machine designers need a PLD with buried, or internal, state registers, compact packaging and high speed. Buried registers free valuable I/O pins; a small package saves precious board space; and high operating speed speaks for itself.

Housed in a 20-pin DIP, the smallest PAL device to date for building state machines sports registers that save I/O pins, yet it is easy to test.

Approaching the ideal is the AmPAL-23S8, the first 20-pin programmable array logic (PAL) device designed specifically for building state machines and sequencers. Its architecture is more flexible than any 24- or 28-pin PAL device or programmable logic array (PLA). For example, it has 14 edge-triggered, D-type flip-flops (Fig. 1). That alone compares favorably with 24-pin bipolar PAL devices, which have up to 10 registers, and with 24- and 28-pin PLAs, which have 12 to 14 registers. Only 40-pin PAL devices with 16 registers surpass the new chip.

As significant is the fact that of the chip's registers, six hold buried state bits. By definition, the buried registers do not merely free I/O pins: Indeed, they actually enhance the chip's value as a state machine. Driven by the chip's AND-OR array the buried state registers, along with the eight registers dedicated to I/O pins, make possible a wide range of states and outputs.

Moreover, the buried state bits can be easily accessed, controlled, and observed. Although the first two attributes are common among PAL devices, the third has always presented problems. The 23S8 is the first chip whose product-term array makes the buried state registers observable.

Of the other eight registers, four serve in output logic macrocells and four store output states. Each macrocell contains three main blocks (Fig. 2). One block is the register, a rising-edge–triggered flip-flop sharing asynchronous reset and synchronous preset inputs with all the other flip-flops. The other two blocks are multiplexers, one of them selecting from one of four output paths, the other from one of two feedback paths.

The logic chip is small, fast, and easy to program. Packaged in a 0.3-in. DIP, it is half the width of 28- and 40-pin PAL and PLA type devices. Moreover, it comes in 33- and 28.5-MHz versions, with propagation delays of 20 and 25 ns, respectively. The faster version is an ideal companion for new 8-MHz, 16-bit microprocessors, as well as for 16-MHz, 32-bit units. Even at 28.5 MHz, the new chip is faster than existing bipolar PLA-based sequencers, which operate from 20 to 25 MHz. It also beats equivalent-density PAL devices operating at or below 25 MHz.

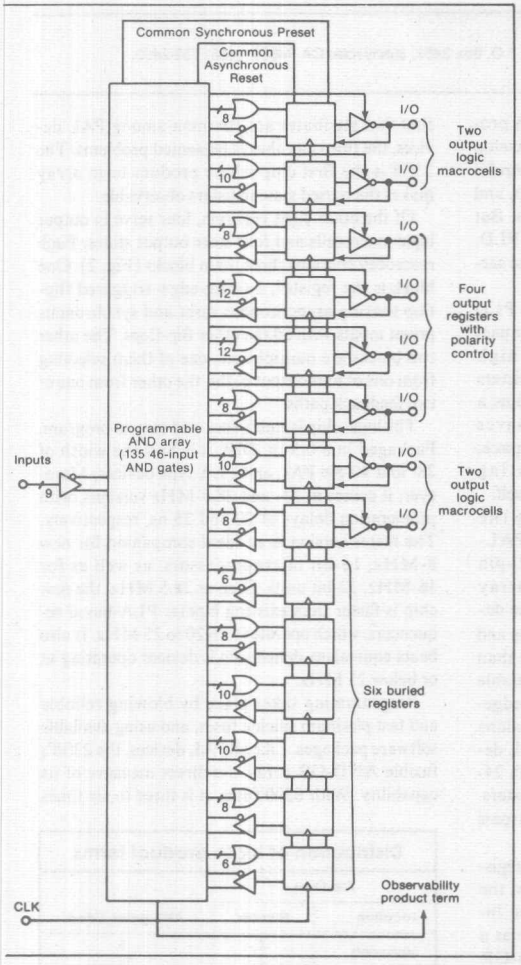
Programming takes place by blowing reliable and fast platinum silicide fuses, and using available software packages. Like all PAL devices, the 23S8's fusible AND-OR array is a direct measure of its capability. With 6200 fuses, it is three to six times

Distribution of logic product terms

Flip-flops		
Location	Number	AND gates ORed
Macrocells	2 2	8 10
Output pins	2 2	8 12
Buried (internal) registers	2 2 2	6 8 10

Reprinted by permission of Electronic Design, July 1986. All rights reserved.

chips, which are three to four times as large and have some 8000 fuses. The new chip's fuse array handles all 14 registers, with 6 to 12 logic products for each output, and up to 23 inputs—9 of them dedicated, 8 for feedback, and



1. The AmPAL23S8 is the first 20-pin PAL device designed specifically for building state machines and sequencers. It contains 14 registers, 6 of them buried and requiring no I/O pins, 4 dedicated to output macrocells, and 4 storing output bits. The states of the buried registers are easily observed, which simplifies debugging.

terms are two other measures of its capability. It has altogether 135 product terms for logic and control functions. Moreover, the AND array that drives the 14 flip-flops is variably distributed (see the table).

The distribution of the 124 logic product terms helps adapt the chip to system requirements, while optimizing its internal resources. (Of the 11 other terms, 8 control output, the remaining 3 control preset, reset, and observability, respectively.) For additional flexibility in designing powerful state machines, the chip can trade off the product terms of its buried registers.

OBSERVABILITY CLOSE UP

The chip's observable product term is generated by the AND-OR array, along with an ability to preload the buried registers. Observability helps check a buried register's state, so that a designer can monitor any register during debugging.

Under control either of a single pin or of a combination of six input signals, the observable product term drives a set of inverting buffers. In normal operation, the buffers also serve the four dedicated output registers and two of the four macrocells. When activated, therefore, the observable product term disables signal flow from those registers and macrocells and simultaneously connects the six buried registers to their respective I/O pins.

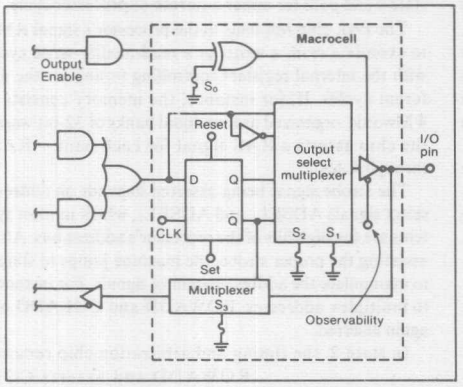
Flexibility in configuring a state machine derives from the independent control of output and feedback multiplexers. Each output macrocell has three fuses, two to determine the output path and one the feedback path. The fuses can configure the macrocells in eight ways: a combinatorial or a registered output with either an I/O pin or a registered feedback. Each of those four variations offers either active high or active low outputs.

In the output path, fuse S_1 determines whether the output is active high or active low. The output-nature fuse, S_2, selects sequential or combinatorial operation. When both of those fuses are intact, a flip-flop's Q pin passes through an inverting buffer, making the output active low and registered.

On the other hand, programming or blowing S_1 renders the output term active high; blowing S_2 causes the output to bypass the register, making it combinatorial. A combinatorial output with feedback can, moreover, provide more than one level of ORing logic. Finally, fuse S_3 controls the feedback path, sending either the flip-flop's Q output or the I/O pin back into the array.

Controlling each output's enable line through a product term from the AND array creates further options by transforming an I/O into a dedicated input, a dedicated output, or a dynamically controlled I/O pin. A designer can thus adjust the number of input and output pins, and have the dynamic I/O capability required by most buses.

Each output enable product term is further associated with an individual polarity fuse. This polarity fuse allows designers to modify a control signal to enable outputs using De Morgan's theorem, which is especially useful for bus control applications.



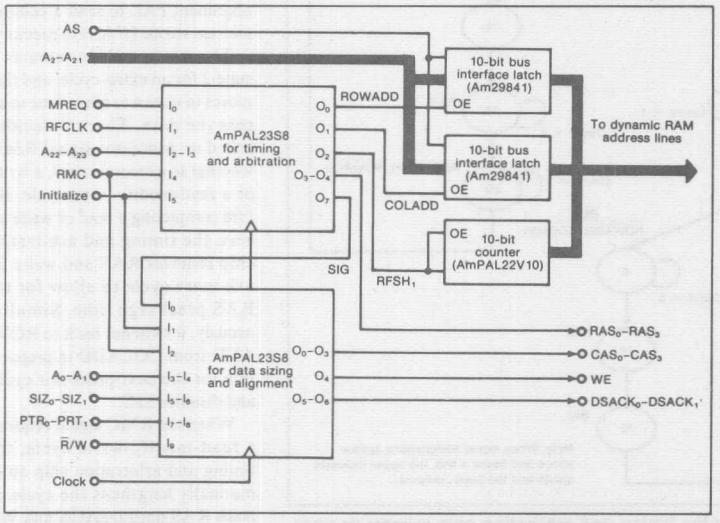
2. Each programmable macrocell has three main blocks: a rising-edge–triggered D-type flip-flop, an output multiplexer, and a feedback multiplexer. The output multiplexer selects from one of four output sources; the feedback multiplexer feeds either an I/O pin or the flip-flop's Q output into the programmable AND-OR array.

The chip's high speed and buried state registers make it an ideal foundation for a flexible and fast dynamic RAM controller that meets the complex requirements of the latest microprocessors. Even at a 16-MHz processor clock rate it can become a controller with few or no wait states that might otherwise have to be built from scratch.

There is, for example, no ready-made controller that can take advantage of dynamic bus sizing — the automatic adjustment to the amount of data movable in one bus cycle. Nor can any off-the-shelf unit handle misaligned data, that is, when an operand falls outside its proper memory boundaries. Bus sizing and data alignment are features of the recently arrived Motorola 68020 and the Intel 80386, both 32-bit microprocessors. But a controller able to perform the two functions can be built around two 23S8s, one for timing and arbitration logic and the other for data sizing and alignment (Fig. 3).

The timing and arbitration chip arbitrates among present, refresh, and CPU cycles; executes read or write, read-modify-write, and refresh cycles; and asserts interface signals. The chip first arbitrates between the processor's Memory Request (MREQ) and Refresh Request (RFCLK) signals. It gives priority to the cycle currently being executed, follows this with the refresh cycle, and concludes with the processor cycle. Both intermittent and burst-mode refresh schemes are therefore possible.

The design's timing relationships and arbitration requirements define a state-machine diagram (Fig. 4). The actual state machine employs five of the six available buried registers. The sixth register is a flag and so acts as a small, independent state machine. Either of two software packages — Cupl from Personal CAD Systems Inc.'s Assisted Technology Division (San Jose, Calif.) or Abel from Data I/O Corp. (Redmond, Wash.) — easily describes the state operations. (Cupl is the language used in this particular example.)



3. Two 23S8s lie at the heart of an advanced dynamic RAM controller that compares well in its data-transfer features—such as dynamic bus sizing and handling misaligned data—with recent 32-bit microprocessors. One of the two main PAL devices controls the timing and arbitration of the memory cycles; the other sizes and aligns data transfers.

No arbitration takes place during the execution of a cycle, which is indicated by a state other than 0. The timing and arbitration chip stores any refresh request and responds to it

later, continuing to follow the current cycle through its various states. Thus the current state automatically takes priority, and no other cycle is started until the current one is complete.

At the end of the cycle, at state 0, the timing and arbitration chip arbitrates between MREQ (a CPU cycle) and RFCLK. In the small independent state machine, the buried register flag, B_5, latches RFCLK, which may last no longer than one clock cycle. When RFCLK is high, B_5 toggles to state 1 and stays there until the refresh cycle is executed. Then it toggles back to state 0 to be ready for the next RFCLK.

During refresh cycles, the timing and arbitration chip asserts the Refresh signal (RFSH), which supplies the refreshed row address to the memory address bus. The chip also generates the Row and Column Address Enable Lines, ROWADD and COLADD, which multiplex addresses onto the memory bus. At state 16, the chip removes ROWADD and asserts RFSH; state 17 asserts the Row Address Strobe signal (RAS) and refreshes all memory banks before removing RAS and RFSH. Removing RFSH increments an external 10-bit refresh-row-address counter (AmPAL22V10) to the next row.

If no refresh request is pending during state 0 and MREQ is asserted, a processor cycle begins and the machine jumps to state 1. Otherwise the machine stays in state 0 and polls for either a refresh or processor cycle.

The PAL chip responds to the processor's signal RMC to execute a read, a write, or a read-modify-write cycle, with the internal registers controlling timing for the different cycles. If, for instance, the memory consists of 4 Mwords, organized as four equal banks of 32-bit words, the chip asserts a RAS signal for each bank—$\overline{RAS_0}$ through $\overline{RAS_3}$.

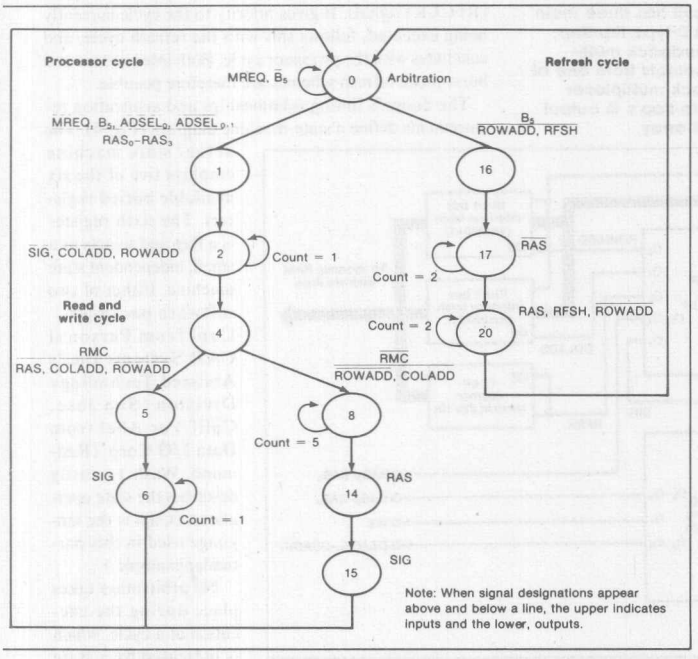
The strobe signal being asserted depends on address-select signals $ADSEL_0$ and $ADSEL_1$, which in most systems are the high bits of the processor's address bus. After asserting the proper strobe, the machine jumps to state 2 to manipulate the address-handling signals. For instance, to multiplex addresses, ROWADD and COLADD are again asserted.

In state 2, the timing and arbitration chip removes ROWADD and asserts COLADD, along with SIG, the reference that tells the data-sizing and alignment chip that a processor cycle is in progress. SIG synchronizes the two 23S8s and instructs the data-sizing and alignment PAL to send a column-address strobe (CAS) to memory.

The state machine remains in state 2 for an extra-cycle, and then moves to state 4 to allow time to access the data. There it decides, based on processor signal RMC, whether to execute a read, a write, or a read-modify-write cycle. Before completing a read or write access, the timing and arbitration chip removes RAS and waits for one more cycle to allow for the RAS precharge time. Simultaneously, it switches back to ROWADD from COLADD in preparation for the next processor cycle, and disables SIG.

When the RMC signal requests a read-modify-write cycle, the timing and arbitration chip automatically lengthens the cycle. It holds RAS until DSACK and WE signals have been asserted, and only then completes the cycle.

The data-sizing and alignment chip works only during a CPU cy-



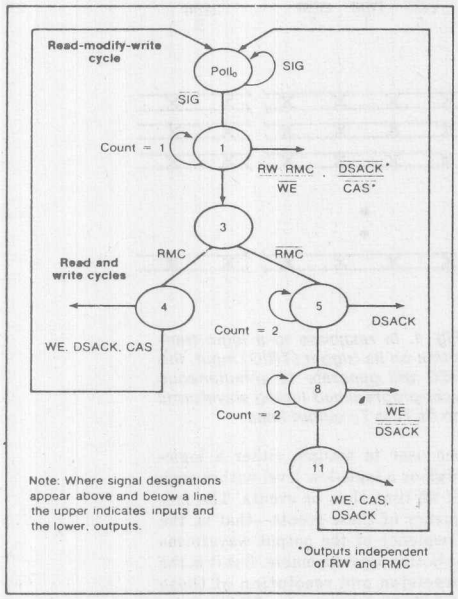
4. A state diagram for the timing and arbitration chip reflects its operation for selecting and orchestrating the memory cycles. The chip waits in state 0 to handle either a processor cycle or a refresh cycle; the latter takes priority over any but a cycle currently in progress. The chip performs refresh, read, write, and read-modify-write cycles.

cle. It creates WE for the write and read-modify-write cycles, and it accepts SIG, which synchronizes the chip's different states (Fig. 5) with those of the timing and arbitration chip.

Four of the data-sizing and alignment chip's buried state registers generate its state-machine function. The machine starts polling in state 0 and stays there until it receives SIG, the start of a processor cycle.

The 68020 has a 32-bit maximum data width and a data bus divided into four byte-wide segments, each independently controlled by a CAS signal from the data-sizing and alignment chip. The processor can therefore read and write byte-wide data over any segment.

The four CAS signals, CAS_0 to CAS_3, strobe the column address from the high- to low-order memory bytes. The signals also allow the microprocessor to read and write across the low-low, low-middle, high-middle, and high-high bytes of a 32-bit data port. The data-sizing and alignment chip must assert all four CAS signals for a 32-bit port; only CAS_2 and CAS_3 are needed to supply a 16-bit port's low- and high-byte strobes; while for an 8-bit port, CAS_3 is enough.



5. The data-sizing and alignment chip works only during a processor cycle and waits in state 0 until one is requested. If the processor calls for a read or a write cycle, the chip asserts DSACK, CAS, and, for a write cycle, WE. For a read-modify-write cycle, it asserts two DSACK signals, one each for the read and write segments.

The data-sizing and alignment chip asserts the CAS signals in state 1, and selects them according to the size of the port in use, the width of the data being moved, and how the data aligns with the port. Coding on input pins PRT_0 and PRT_1 shows the port size; signals SIZ_0 and SIZ_1 from the microprocessor show data width; and the microprocessor's least-significant address bits, A_0 and A_1, give the alignment.

THE PROCESSORS PREFERENCE

The size of the port dictates the amount of data that can be transferred in any one cycle. Based on the PRT signals the state machine asserts DSACK signals, which acknowledge the CPU and tell it how much data it can move, since it will always try to move as much as possible. The state machine synchronizes the acknowledgment timing, asserts the required CAS signals, and holds them until the read, write, or read-modify-write cycle is complete.

A read cycle's DSACK signal is asserted in state 1 and remains until the end of the cycle at state 4. A write cycle is identical, except for the additional WE signal. But a read-modify-write cycle requires two DSACK signals, one for reading, the other for writing. The first DSACK is the same as that for a read or a write cycle; the second DSACK occurs in state 8 and remains until state 11, with a delayed WE asserted in state 8 by an internal 4-bit counter. □

Om Agrawal is product planning manager for programmable logic devices at Advanced Micro Devices. He has designed 16-bit and 32-bit minicomputers and is co-author of a book on high-speed memory systems. He holds a PhD in electrical engineering and computer science from Iowa State University and an MBA from the University of Santa Clara.

Kapil Shankar is a senior product planning engineer for programmable logic and memory devices, and has designed advanced graphics systems. He has an MS in computer and systems engineering and an ME in electrical power engineering, both from Rensselaer Polytechnic Institute.

Programmable Event Generator Conquers Timing Restraints

Bruce Threewitt, Manager, Product Planning
Advanced Micro Devices, Sunnyvale, CA

When it comes to generating complex, high-resolution digital waveforms, Advanced Micro Devices' Am2971 picks up where older parts leave off. Precise time delays once required the use of either a hybrid structure consisting of an analog delay line combined with digital logic or costly counters driven by high-frequency clocks. On the one hand, analog delay lines have short enough tap-to-tap time delays, or resolutions, to be usable for handling the tighter timing relationships of dynamic and static RAMs. However, those lines cannot also accommodate the longer total delays needed by these devices.

On the other hand, the desired waveforms can be obtained digitally from the outputs of one or more binary counters. But to achieve resolutions of 10 ns, counters must be driven with 100-MHz clock frequencies, which are difficult to distribute around a board.

The *programmable event generator* (PEG) a monolithic IC, offers a more elegant solution for the problem of generating complex high-speed timing waveforms (see *Fig. 1*). Applications for the PEG (see *box*, "A new PEG in the designer's tool box") range from simply correcting the clock skew that results from distributing a clock signal on a backplane to generating complex state-machine timing. The PEG's 12 output lines can be programmed by the user to assume either a logic-high or a logic-low level within each of 32 time slots or events. The frequency of these events—that is, the frequency of the output waveforms—is also programmable. But it is the precision and resolution of these waveforms that is the PEG's claim to fame (see *Fig. 2*).

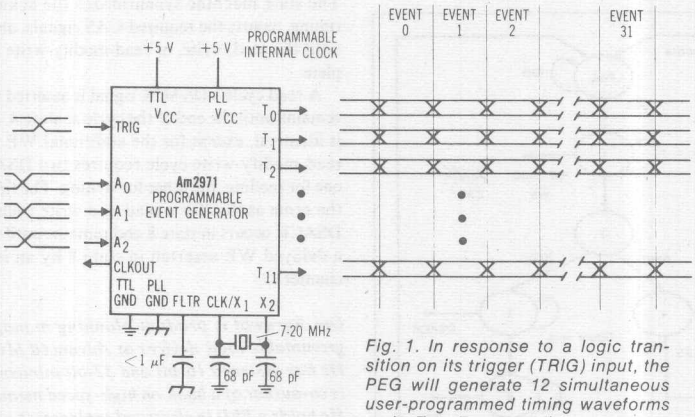


Fig. 1. In response to a logic transition on its trigger (TRIG) input, the PEG will generate 12 simultaneous user-programmed timing waveforms on its T_0 to T_{11} output lines.

Internal operation

The PEG is an edge-triggered logic device that generates a preprogrammed digital waveform in response to a triggering signal. It consists of four basic blocks: a next-address and event store, a start-ad-

Reprinted by permission of Electronic Products, July 1, 1986.

dress store, an oscillator and clock-control block, and control logic (see *Fig. 3*). A total of 623 user-programmable platinum-silicide fuses, similar to those used in AMD's bipolar PROMs and programmable array logic devices, are located throughout these blocks.

When the PEG's fuses are being programmed, the 12 timing-tap outputs (T_0 to T_{11}) are used as fuse-address inputs. A thorough description of the procedure for programming the PEG can be found on its data sheet, so for the purpose of understanding what goes on inside the chip, let's assume that it has already been programmed.

An output-timing sequence is initiated by a logic transition at the TRIG input. Two user-programmable fuses are located in the trigger-polarity block. One of them is a polarity-select fuse, which when unprogrammed will cause the timing sequence to start during a negative transition of the TRIG input. If it is programmed, the sequence starts when a positive transition occurs.

The second fuse in this block is used to define the end of a timing sequence. If this fuse is left unprogrammed, the timing sequence is stopped on the trailing edge of TRIG pulse. Otherwise if the second fuse is programmed, the end of the timing sequence is defined by the stop bits as programmed by the user into the next-address/event generator functional block.

Customized waveforms

A transition at the TRIG input latches three address bits, A_0 to A_2, which are decoded by the start-address store, which serves as an 8 x 5-bit mapping PROM. Each of its eight addresses represents a different starting point for the 12 output waveforms. For example, three different addresses in this store could be used to initialize output waveforms that implement the read-, write-, and page-mode timing sequences of a dynamic RAM.

The five bits of each address are used to select one of thirty-two 18-bit locations from the next-address and event store (also a PROM). Each 18-bit string contains three groups of information. The first 5 bits define the next address of the desired timing sequence. The next 12 bits define the logic levels on each of the timing-output lines (T_0 to T_{11}). The 18th bit is a sequence-stop bit. As the timing sequence progresses from the first to the thirty-second event-store address, the output lines will produce 12 independent waveforms.

The assumption so far is that the

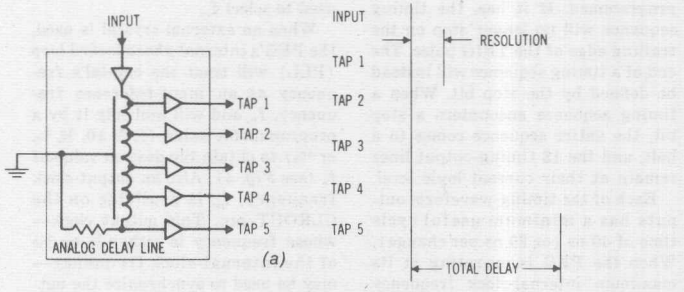
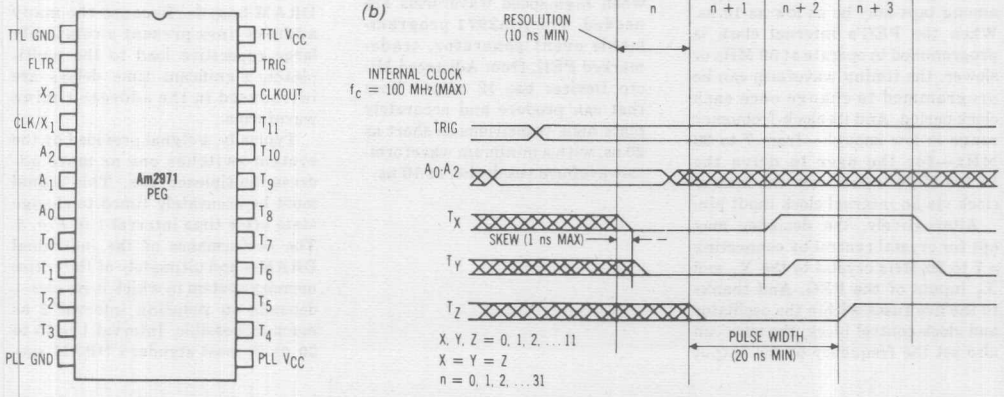


Fig. 2. Tap-to-tap resolution of an analog delay line (a) is not independent of its total delay time. However, the PEG's 12 programmable output waveforms (b) are independent of one another. Resolutions down to 10 ns can be easily obtained.



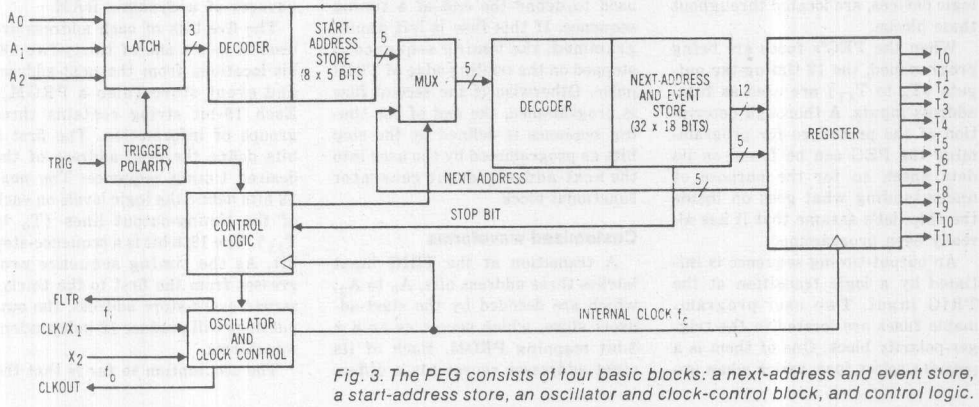


Fig. 3. The PEG consists of four basic blocks: a next-address and event store, a start-address store, an oscillator and clock-control block, and control logic.

stop-function fuse has not been programmed. If it has, the timing sequence will no longer stop on the trailing edge of the TRIG pulse. The end of a timing sequence will instead be defined by the stop bit. When a timing sequence encounters a stop bit, the entire sequence comes to a halt, and the 12 timing-output lines remain at their current logic level.

Each of the timing-waveform outputs has a minimum useful cycle time of 40 ns (or 20 ns per change). When the PEG is operating at its maximum internal-clock frequency (f_c = 100 MHz), the outputs must be programmed to remain unchanged for at least two clock periods for each change of logic level. That is, although an output can only change a minimum of every 20 ns, the timing resolution between events among taps may be as low as 10 ns. When the PEG's internal clock is programmed to operate at 50 MHz or slower, the timing waveform can be programmed to change once each clock period. And its clock-frequency range is low enough—from 7 to 20 MHz—for the user to drive the PEG's CLK input from the system clock via an internal clock input pin.

Alternatively, the designer may opt for crystal control by connecting a 7 to 20-MHz crystal to the X_1 and X_2 inputs of the PEG. And thanks to the five fuses within the oscillator and clock-control block, the user can also set the frequency of the output waveforms. Four of these fuses are used to select f_c.

When an external crystal is used, the PEG's internal *phase-locked loop* (PLL) will treat the crystal's frequency as an input-reference frequency, f_i, and will multiply it by a programmable value (1, 5, 10, ½, ¾, or 1¼) to obtain the desired value of f_c (see *Fig. 4*). Also an output-clock frequency, f_o, is available on the CLKOUT pin. This output clock—whose frequency is either ⅛ or 1/10 of the internal-clock frequency—may be used to synchronize the outputs of the PEG with the remainder of the system. And speaking of systems, the PEG comes in very handy for generating the specialty cycles offered by today's DRAMs.

A new PEG in the designer's tool box

Designers no longer have to grapple with low-resolution analog delay lines or costly timing logic when high-speed waveforms are needed. The Am2971 programmable event generator, trademarked PEG, from Advanced Micro Devices has 12 output lines that can produce and accurately place logic transitions as short as 20 ns, with a minimum waveform-to-waveform resolution of 10 ns.

DRAM-timing application

Typically, the system designer must use more than one analog delay line and some external logic to produce the multiple $\overline{CAS}$ pulses needed for the page-mode cycles of some DRAMs. Timing designs using high-speed clocks and counters are equally cumbersome in this case. Once again, the PEG chip offers a streamlined solution because it can readily generate the multiple $\overline{CAS}$ pulses.

Another job for the PEG comes about as a result of the use of multiplexed address lines on high-density DRAM boards. Because the many address lines present a relatively large capacitive load to the multiplexer, significant time delays are introduced in the address-timing waveforms.

Typically, a signal provided by the system switches one or more address-multiplexer ICs. This signal must be accurately timed to change state after time interval 1 in *Fig. 5*. The performance of the individual DRAM—and ultimately of the entire memory system in which it resides—depends on reducing interval 2 as much as possible. Interval 1 is 15 to 20 ns in most standard DRAM spe-

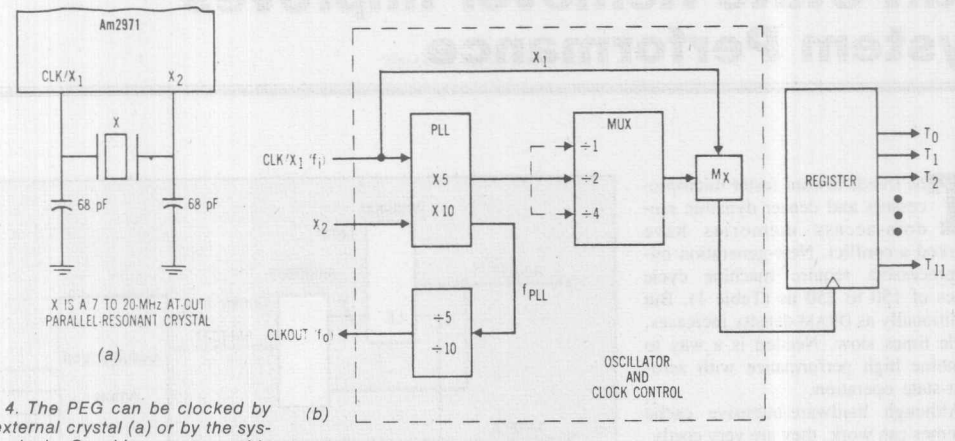


Fig. 4. The PEG can be clocked by an external crystal (a) or by the system clock. On-chip programmable clock-control logic (b) enables the user to set the internal-clock frequency, f_C.

cifications. Thus, a resolution of 10 to 15 ns is desirable for the DRAM-timing waveforms—an easy job for the PEG.

What's more, the PEG can be used to generate the necessary timing waveforms for multi-array memories too. In video-DRAM systems, for example, the rising edge of the transfer-enable signal, $\overline{TRG}$, must be accurately placed relative to the rising edge of the serial-port clock signal, SC. When the PEG is used to generate both waveforms, the designer has full control over the resolution between the rising edges of the $\overline{TRG}$ and SC signals.

More events and/or channels

In some applications, 12 output channels are sufficient but more than 32 event states are necessary. These additional states can be obtained by connecting two PEGs as shown in *Fig. 6*. After PEG 1 has cycled through its 32 states, its T_0 line triggers PEG 2. The second PEG will then begin its timing sequence at the start address specified by PEG 1. If necessary, the clock frequency of PEG 2 may be changed by using the clock output of PEG 1 as the clock input to PEG 2. Otherwise, the internal clocks and TRIG polarities of the two PEG chips are independently programmable.

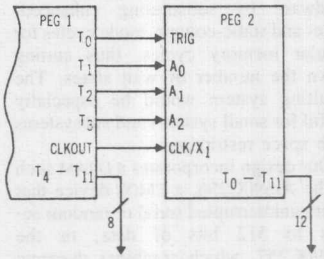


Fig. 6. By connecting two PEGs, designers can obtain timing waveforms that consist of more than 32 events.

If the waveforms must be nested, one of the output taps of PEG 2 can be used to remove the TRIG signal from PEG 1. PEGs can also be paralleled for applications in which 32 event states are sufficient but more than 12 output channels are needed. For example, when 12 to 24 channels are needed, the designer can simply drive two PEG chips with the same input signals and use as many of the output lines as necessary. □

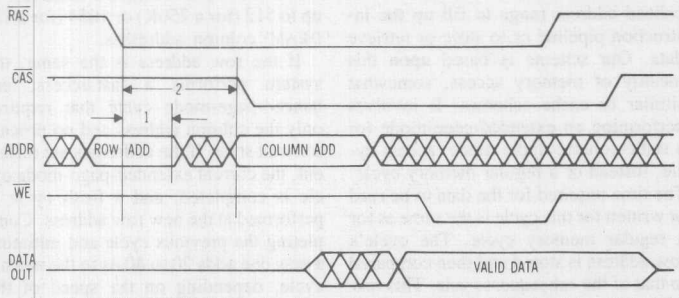


Fig. 5. In this DRAM delayed-write cycle, interval 1 must be kept to within 10 or 15 ns, and interval 2 must be as short as possible. When mounted on the same DRAM memory board, the PEG could be used to generate the $\overline{RAS}$, $\overline{CAS}$, and $\overline{WE}$ signals.

System Performance

The trends toward faster microprocessors and denser dynamic random-access memories have sparked a conflict. New-generation microprocessors require machine cycle times of 150 to 250 ns (Table 1). But traditionally as DRAM density increases, cycle times slow. Needed is a way to combine high performance with zero-wait-state operation.

Although hardware-intensive cache schemes can work, they are very costly. It is possible, however, to improve system performance with little additional hardware by substituting enhanced-page- and static-column-mode cycles for regular memory cycles, thus cutting down the number of wait states. The resulting system would be especially useful for small systems and subsystems with space restrictions.

Our design incorporates a DRAM such as the Am90C256, a CMOS device that offers uninterrupted serial or random access to 512 bits of data, or the Am90C257, which combines dynamic storage and static decode to nearly double sequential access rates. These devices provide extended-page and static-column access modes at a small increase in cost. These modes permit data access times of 40 to 60 ns, enabling the memory controller to remove wait states for several processor cycles.

The only additional hardware is an AmPAL22V10, a 24-pin programmable logic device that offers an AND/OR logic structure for custom programming, 22 inputs, and 10 outputs with 12 product terms each to provide timing and arbitration for extended-page-mode and refresh cycles. Also needed are two devices to latch and compare row addresses and a counter to track the extended-page-mode cycles' time out.

Usually, microprocessors perform a number of continuous accesses in a localized address range to fill up the instruction pipeline or to store or retrieve data. Our scheme is based upon this locality of memory access, somewhat similar to cache schemes. It involves performing an extended-page-mode (or a static-column/ripple-mode) access cycle, instead of a regular memory cycle. The time required for the data to be read or written for this cycle is the same as for a regular memory cycle. The cycle's row address is stored and then compared to that of the subsequent cycle. This row address would be the same for a localized access cycle within the same row of up to 512 (for a 256K) or 1054 (for a 1M DRAM) column addresses.

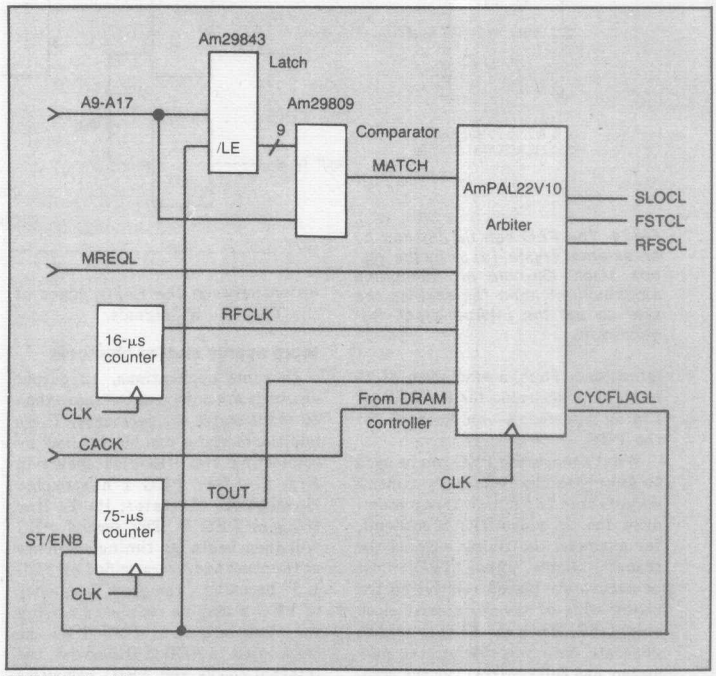


FIGURE 1. Wait state remover arbitration scheme.

If the row address is the same, the system performs a fast-access, enhanced-page-mode cycle that requires only the column address and no processor wait state. If the addresses are different, the current extended-page-mode cycle is completed, and a fresh cycle is performed at the new row address. Completing the previous cycle and initiating a new one adds 20 to 40 ns to the normal cycle, depending on the speed of the DRAM controller. Usually, one wait-

Reprinted with permission of *VLSI*,
November 1986, Vol. VII, No. 11

state processor cycle provides additional time of 50 to 125 ns, enough to squeeze in this overhead and not cause further wait states.

Because row addresses can be compared during the same time interval as the address decoding and generation of a memory request or select, this action does not increase cycle time. The result is a regular cycle speed with one (or two) wait states for random memory accesses at different row addresses and fast, no-wait-state cycles for localized accesses at the same row address. Depending on the program, execution speed can increase 5% to 20%.

The AmPAL22V10 also acts as a timer and arbiter for the extended-page-mode and refresh cycles (Figure 1 and Table 2). Its refresh-cycle counter runs at 16 μs. But the DRAM controller does not perform a refresh every time it receives a request: because the page-mode cycle can be extended up to 75 μs to allow the most CAS-only fast cycles, refreshes at 16-μs intervals would disrupt operation. Instead, the arbitration scheme is designed so that, during an extended-page-mode cycle, the memory controller counts up to four refresh requests before performing the refresh cycles.

Description of Circuit And Boolean Logic

For the sake of clarity, the circuit diagram shows a general-purpose arbitration interface, not a complete DRAM controller. It can be either integral to a PAL-based DRAM controller or linked to a VLSI DRAM controller, depending on the user's needs.

For 256K DRAMs, an Am29843 with 9-bit latches stores a 9-bit row address. The arbiter PAL controls the latch-enable signal, $\overline{LE}$. A 9-bit Am29809 compares the stored row address with that of the current cycle directly from the processor on address lines A9-A17. If the addresses match, the Am29809 sends a MATCH signal to the arbiter PAL. The arbiter PAL also receives the memory-request ($\overline{MREQ}$) signal from the address-decoding circuitry (not shown here).

Of the AmPAL22V10's two counters, the refresh counter is free running and generates refresh requests (RFCLK) at fixed 16-μs intervals. The counter for the extended-page-mode cycle is controlled by the arbiter PAL, which starts the counter by sending a signal (CYCFLAG). On completion of the 75-μs timing count, the counter sends a time-out signal, TOUT, to the arbiter, forcing it to complete the current cycle and remove the CYCFLAG signal. If the arbiter requires a new extended-page-mode cycle at a different row address before this timeout, it resets the counter by removing the CYCFLAG signal. So the counter starts when CYCFLAG is high and stops or resets when CYCFLAG is low.

The arbiter's operation is quite simple. It uses registers SLOC, FSTC, and RFSC as flags to indicate a cycle request to the memory controller. SLOC requests the initiation of an extended-page-mode

Processor	Cycle time (ns)		
	0 wait	1 wait	2 waits
8088/8086 (8 MHz)	500	625	750
8088/8086 (10 MHz)	400	500	600
80286 (8 MHz)	250	375	500
80286 (10MHz)	200	300	400
80386 (8 MHz)	188	250	313
68020 (16 MHz)	188	250	313
68020 (20MHz)	150	200	250

TABLE 1. Microprocessor cycle times.

```
INPUTS
        MREQ
        MATCH
        TOUT
        RFCLK
        CACK
OUTPUTS
        CYCFLAG
        SLOG
        FSTC
        RFSC
        RCT0
        RCT1
/CYCFLAG:  = /MREQ * CYCFLAG * RCT1 * RCT0                ; Assert for R/W
             + /CYCFLAG * /TOUT * (RCT1 + RCT0)           ; Unassert based
             * (MREQ + MATCH)                             ; on 3 conditions
/SLOC:     = /MREQ * CYCFLAG * SLOC * FSTC                ; Start extended
             * RFSC * RCT1 * RCT0                         ; cycle
             + /MREQ * /CYCFLAG * /MATCH * SLOC           ; Row address not
             * FSTC * RFSC * RCT1 * RCT0                  ; matched new
                                                          ; extended cycle
             + /SLOC * CACK                               ; Hold till complete
/FSTC:     = /MERQ * /CYCFLAG * MATCH * SLOC              ; Match CAS only
                                                          ; fast cycle
/RCT1:     = (RCT1 * /RCT0 + /RCT1 * RCT0)                ; Count up
             * RFCLK
             + CT * RFSC                                  ; Count down
             * (/RCT1 * /RCT0 + RCT1 * RCT0)
             + /RFCLK * (/CT + /RFSC * /RCT1              ; Hold count
/RCT0:     = RFCLK * RCT0                                 ; Count up
             + CT * RFSC * RCT0                           ; Count down
             + /RFCLK * (/CT + /RFSC) * /RCT0             ; Hold count
CT:        = /RFSC
```

TABLE 2. Listing of equations for arbiter.

cycle. This new cycle continues, even after completion of a data transfer, as long as CYCFLAG is high. FSTC requests the initiation of a fast CAS-only cycle at the same row address at which an extended-page-mode cycle is in progress, meaning that CYCFLAG must be high for the FSTC signal to be asserted.

The third RFSC requests the initiation of a refresh cycle. All cycle requests for new slow extended-page-mode cycles, fast CAS-only extended-page-mode cycles, and refresh cycles are arbitrated on the basis of inputs MREQ and MATCH, and status feedbacks of CYCFLAG and (RCT0, RCT1), a counter to keep track of number of refresh requests. A new cycle request is arbitrated only when no current cycle request is active, indicated by SLOC, FSTC, and RFSC being low.

When CYCFLAG is low, the DRAM controller should complete the existing extended-page-mode cycle (usually by deactivating both RAS and CAS signals to the memory). As shown in the Boolean equations, CYCFLAG goes low if at least one of the following three conditions is encountered:

1. Extended-page-mode cycle timeout (TOUT) high.
2. Refresh count up to four (RCT0 = 1 and RCT1 = 1).
3. New cycle at a different row address MREQ and MATCH.

This allows arbitration to begin afresh for a new cycle. Because CYCFLAG is generated for each new extended-page-mode cycle, it is also used as a latch-enable signal for the 9-bit row-address latch.

The counter made by register RCT0 and RCT1 remembers refresh requests made during an extended-page-mode cycle. It counts up by one upon receiving each RFCLK, to the maximum of four; it counts down by one when a refresh cycle is executed. An extra register, CT, monitors refresh-cycle execution by detecting the rising edge of the refresh-cycle request RFSC. CT and RFSC low indicate the refresh cycle's completion, and this is used to decrement counter RCT0, RCT1. The refresh cycles are repeated, and no memory cycles are allowed, until this counter is reset to zero. □

PLDs Implement Encoder/Decoder for Disk Drives

By using software to define programmable logic devices as run-length-limited encode/decode systems, you can design disk-drive systems that have 50% more data-storage capacity than drives that implement the MFM code.

Arthur Khu and Rudy Sterner,
Advanced Micro Devices Inc

When you're designing a disk-drive system, you can implement run-length-limited (RLL) 2,7 encoding/decoding circuitry in your design by using only three programmable-logic devices (PLDs) and two shift registers. You design the encoder and decoder as state machines and use the timing diagrams to determine what the timing and control signals must be.

To create a disk controller with encoding/decoding features, you can use three AmPAL22V10 PLDs and a disk controller such as the Am9580/Am9582 chip set (**Fig 1**). The Am9580 hard-disk controller and the Am9582 disk-data separator perform all the general disk-control functions. The PLDs have appropriate architectures for implementing the encoding and decoding state machines. Further, you can reprogram the PLDs to implement higher density ratios for data encoding, and you can increase your system's speed simply by using faster PLDs.

An RLL code is a code in which the number of zeros between ones—the run length—is definite. The "2,7" designation means that the code for the binary data string has at least two and at most seven zeros separating the ones. RLL codes increase the density of data stored on a disk by reducing the number of recorded pulses (ones) necessary to represent a given amount of data. This reduction allows the disk-drive circuitry to pack the ones closer together, increasing the amount of data on the disk.

In comparison with the (de facto) industry-standard approach, MFM, the RLL 2,7 code increases by 50% the amount of data you can store on a disk drive (see **box,** "RLL 2,7 code vs MFM code"). In addition, RLL

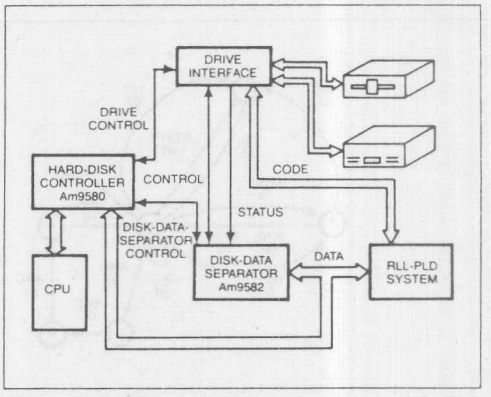


***Fig 1—A complete disk controller** requires only two VLSI ICs, a PLD-based encoding/decoding system, and a drive interface. The hard-disk controller and the disk-data separator provide generic disk-drive control, and the PLDs provide RLL 2,7 encoding and decoding, which increases disk storage capacity.*

Reprinted by permission of EDN, September 18, 1986.

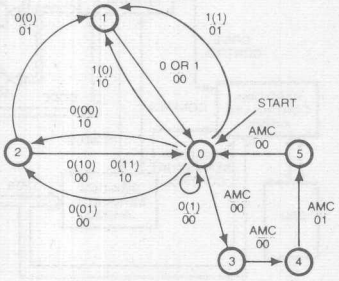
TABLE 1 — RLL 2,7 CODING RULES

DATA	2,7 CODE
10	1000
11	0100
000	100100
010	001000
011	000100
0010	00001000
0011	00100100

2,7 decoding circuitry recovers quickly from code-detection errors.

As **Table 1** shows, RLL 2,7 encoding circuitry translates seven data strings into 2,7 code. You can break any binary non-return-to-zero (NRZ) data string into combinations of the seven data strings. To obtain the decoded data string, the circuitry matches the 2,7 code patterns with the seven 2,7 code strings in **Table 1.**

Because 2,7 code strings have variable lengths (they can be 2, 3, or 4 bits long), your design will need control logic that controls the output from the encoder/decoder as translation takes place. Encoding and decoding state machines (**Figs 2** and **3**) implement this control logic from the code in **Table 1** (see **box,** "Convert RLL 2,7 code to a state machine"). The encoding state machine



KEY:
DATA BIT TO CODE
0(01) ← 2 LOOK-AHEAD BITS
00 ← 2,7 CODE FOR DATA BIT 0
AMC ← IF ADDESS MARK
00 ← SIGNAL HIGH, 2,7 CODE PRODUCED

(a)

NOTE: STATES 0, 3, 4, AND 5 ARE USED FOR ADDRESS-MARK GENERATION

TO CODE 00101100 . . .

CODE CYCLE	T_0	T_1	T_2
CURRENT STATE	0	2	0
CONTENTS OF 4-BIT INPUT SHIFT REGISTER INREG	BIT TO CODE SERIAL OUT ← 0 0 1 0 ← SERIAL IN 2-BIT LOOK-AHEAD SHIFT IN A NEW BIT BEFORE GOING TO CYCLE T_1	BIT TO CODE SERIAL OUT ← 0 1 0 1 ← SERIAL IN 2-BIT LOOK-AHEAD SHIFT IN A NEW BIT BEFORE GOING TO CYCLE T_2	BIT TO CODE 1 0 1 1 ←
2, 7 CODE TO LOAD INTO OUTREG	00	00	
NEXT STATE	2	0	

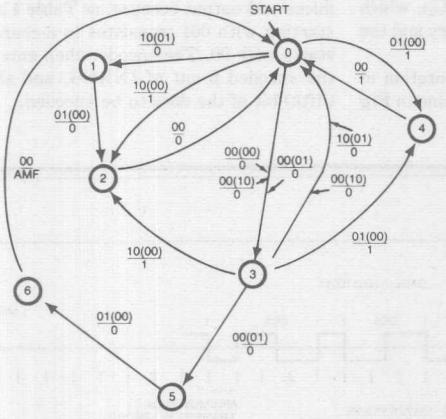
(b)

Fig 2—The encoder state machine ***(a)*** *describes the translation of input data into RLL 2,7 code. The first two cycles in the encoding of the data string 00101100* ***(b)*** *demonstrate how the encoder determines the correct code by examining both the bit to be encoded and the next two (look-ahead) bits.*

During decoding, circuitry matches the RLL 2,7 code patterns with the seven RLL 2,7 code strings to obtain the corresponding decoded data string.

produces two 2,7 code bits for each data bit received. The decoding state machine, on the other hand, produces one data bit for every two bits of 2,7 code. The clocking scheme in these encoding/decoding state machines is simpler than the familiar table-look-up method, which requires suspended operation during encoding and decoding.

You can implement both of these encoding/decoding 2,7 state machines with one PLD device (IC_1) and two shift registers (INREG and OUTREG) (**Fig 4**). To



KEY:
2 CODE BITS TO DECODE
10(00) → 2-CODE-BIT LOOK-AHEAD
1 ← DECODED DATA BIT

00 ← 2 CODE BITS TO DECODE
0 DECODE; NO NEED FOR LOOK-AHEAD
DECODED DATA BIT

00 ← 2 CODE BITS TO DECODE
AMF ← ACTIVATE ADDRESS-MARK-FOUND SIGN

(a)

NOTE: STATES 0, 3, 5, AND 6 ARE USED WHEN DETECTING ADDRESS MARKS

TO DECODE 00100010 . . .

DECODE CYCLE	S_0	S_1	S_2
CURRENT STATE	0	3	2
CONTENTS OF 4-BIT INPUT SHIFT REGISTER INREG	2 CODE BITS TO DECODE SERIAL OUT ← 0 0 1 0 ← SERIAL IN 2-CODE-BIT LOOK-AHEAD SHIFT IN TWO NEW CODE BITS BEFORE GOING TO CYCLE S_1	2 CODE BITS TO DECODE SERIAL OUT ← 1 0 0 0 ← SERIAL IN 2-CODE-BIT LOOK-AHEAD SHIFT IN TWO NEW CODE BITS BEFORE GOING TO CYCLE S_2	2 CODE BITS TO DECODE 0 0 1 0
DECODED DATA BIT TO LOAD INTO OUTREG	0	1	
NEXT STATE	3	2	

(b)

Fig 3—This decoder state machine (a) *represents the translation of RLL 2,7 code into an output data stream. To determine the correct output data, the decoder examines four code bits—two bits to be decoded and two look-ahead bits, as shown in the example in* ***b.***

Because the RLL 2,7 code contains two bits for every one bit of data, the timing circuitry divides the clock signal for the coded data, producing a data clock.

synchronize the encoder/decoder with the hard-disk controller, you use two other PLDs (IC_2 and IC_3) to provide clock-generation and address-mark-control logic. IC_1 is a AmPAL22V10, which has sufficient capacity to implement the two state machines. IC_2, also an AmPAL22V10, utilizes the PLD's large capacity and programmable output cells to implement the address-mark-control circuitry. An AmPAL16HD8 (IC_3), which is sufficient for implementing the clock circuitry and the random logic, completes the design.

To understand the state-machine implementation of the RLL 2,7 encoder, consider the state machine in **Fig 2.** The encoding state machine begins in state zero; the first four bits of the data to be encoded are in the shift register INREG. For the data stream in the figure, the encoder, beginning in the first cycle (T_0) reads the first bit in the stream as 0 and sees that the next two bits (the look-ahead bits) in INREG are 0 and 1, respectively. According to the state diagram, the encoder produces a 00 output because, as **Table 1** shows, input data starting with 001 translates to a character string that starts with 00. The encoder then enters state 2, shifts the encoded 0 out of INREG, and shifts in the next (fifth) bit of the data to be encoded.

RLL 2,7 code vs MFM code

Although the RLL 2,7 and MFM coding methods can both increase a disk drive's capacity, RLL 2,7 code is more compact. A disk drive that implements the RLL 2,7 code can, therefore, store 50% more data than can a drive that implements the MFM code.

On the magnetic medium in the disk drive (hard-disk or floppy-disk drives), binary data appears as a change in flux (representing a one) or as no change in flux (representing a zero). Because the disk density is limited by the minimum distance between flux transitions, the maximum data density depends on how the data is encoded.

MFM is an RLL code with the designation 1,3;1,2;1. RLL 2,7 code (its full designation is 2,7;1,2;3) allows a minimum of two zeros between each one, and MFM code allows a minimum of one zero. RLL 2,7 code can store as much as 50% more data in a given number of flux changes than can MFM code; therefore, RLL 2,7 code can store as much as 50% more data on a given section of magnetic medium.

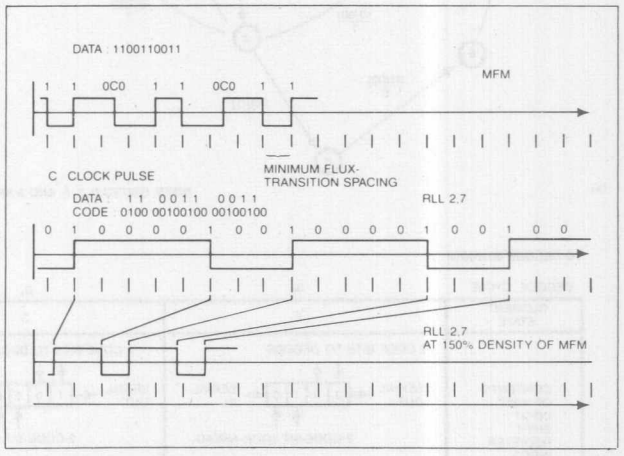


***Fig A—Data storage that's 50% more dense** is the principal advantage of RLL 2,7 code over the de facto standard MFM code. Because it needs fewer transitions to describe data, the RLL 2,7 code takes up only 67% of the disk storage space that MFM code occupies.*

Consider the example in **Fig A.** The data stream 1100110011 is represented by eight transitions in MFM code, and by five transitions in RLL 2,7 code. When the disk drive uses the minimum distance between transitions to record the RLL 2,7 code, the RLL 2,7 code takes 67% of the space that the MFM code takes, so it has space available to store 50% more data. In most applications, the actual storage increase is between 35% and 40%, because some disk space is reserved for sector and data-field markers.

In the second cycle (T_1) of the encoding process, the encoder sees that the data bit is 0 and the two look-ahead bits are 1 and 0, respectively. According to the state diagram, when the encoder is in state 2 and sees 010, its output is 00. As before, the encoder then moves to the next state (in this case, state 0) and shifts a new bit into INREG. The rest of the encoding process proceeds similarly.

The decoding process is similar to the encoding process. The decoder, which is described as a state machine (**Fig 3**), accepts two input bits in RLL 2,7 format and sees the next two coded bits as look-ahead bits. In contrast to the encoder, the decoder produces one output bit for every two input bits.

To implement this encoder and decoder circuitry with PLDs, you can use PLD-design tools such as CUPL from Personal CAD Systems Inc (San Jose, CA) and Abel from Data I/O Corp (Redmond, WA). These software tools include syntaxes that you can use to describe state machines as well as general logic equations.

To control the rate at which IC_1 (in **Fig 4**) receives data and code, IC_2 and IC_3 implement the timing signals shown in **Fig 5**. Three signals (Read, Write, and Code_Clock) from the hard-disk controller and disk-data separator form the basis of the timing signals.

First, the timing circuitry produces a clock signal, Rd_Clk, from the Code_Clock signal that originates at the disk-data separator's FDDAM output. Because the RLL 2,7 code contains two bits for every one bit of

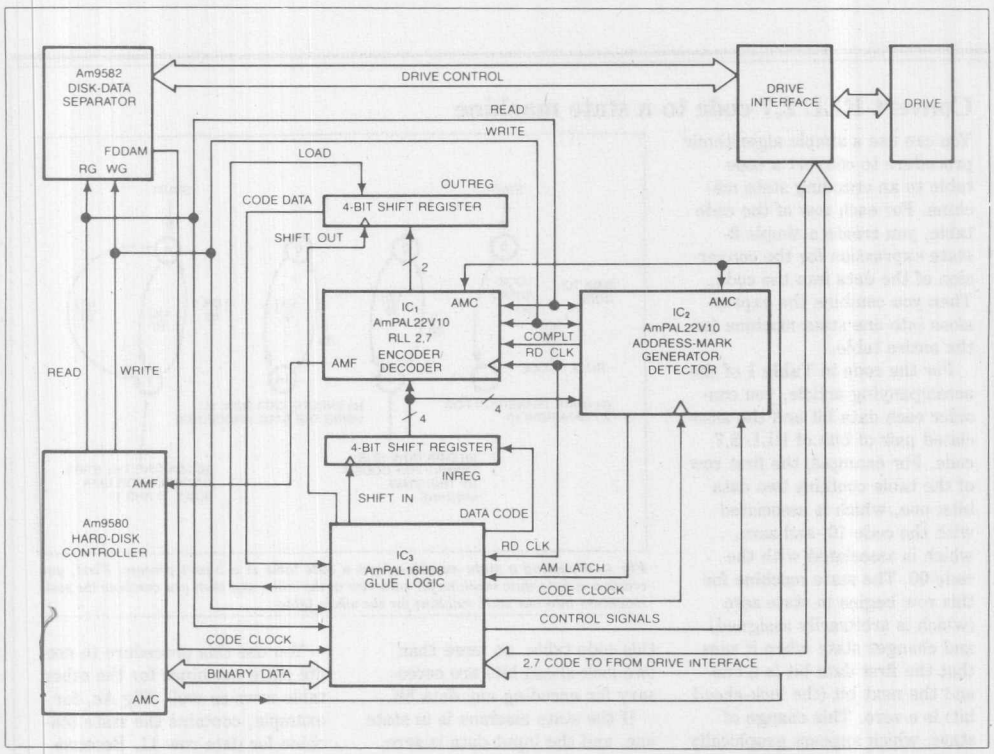


Fig 4—The encoding/decoding circuitry *includes three PLDs. IC_1 implements the encoding and decoding state machines. IC_2 monitors IC_1 and provides special timing and input signals. IC_3 implements glue logic and timing signals.*

Before the disk-drive system can decode data on the disk, it must synchronize itself with the disk's data clock to find out when the data bits begin.

data, the timing circuitry divides the clock signal for the coded data (Code_Clock), producing a data clock signal, Rd_Clk. The timing circuitry then uses Code_Clock and Rd_Clk to control the timing of shift-register control signals Shift_Out and Shift_In.

To obtain the equations you need to program the PLDs, examine the timing diagrams. The timing diagrams show that the state-machine design requires only a few timing signals to implement the controller. Because the timing circuitry loads and shifts data produced by IC_1 at the points indicated on the timing diagram in **Fig 5**, the only clock signal that IC_1 requires to code or decode data in INREG is the Rd_Clk signal.

To control the transfer of data and code, the Shift_In signal latches data or code into INREG, and the Shift_Out signal controls the output of OUTREG. For example, when encoding data, the encoder produces two bits of code on each rising edge of Rd_Clk. Because

Convert RLL 2,7 code to a state machine

You can use a simple algorithmic procedure to convert a code table to an encoding state machine. For each row of the code table, you create a simple 2-state expression for the conversion of the data into the code. Then you combine the expressions into one state machine for the entire table.

For the code in **Table 1** of the accompanying article, you consider each data bit and the associated pair of bits of RLL 2,7 code. For example, the first row of the table contains two data bits: one, which is associated with the code 10, and zero, which is associated with the code 00. The state machine for this row begins in state zero (which is arbitrarily assigned) and changes state when it sees that the first data bit is a one and the next bit (the look-ahead bit) is a zero. This change of state, which appears graphically in **Fig Aa**, results in the output 10.

Note that the state machine must evaluate the look-ahead bit. If this bit is a one instead of a zero, the input data will be 11, corresponding to the second row of the table, so your state machine must generate a 01. For this code table, no more than two look-ahead bits are necessary for encoding any data bit.

If the state machine is in state one, and the input data is zero, the state machine produces a 00 output and returns to state zero (**Fig Ab.**) Note that the data bit encoded in this state is the look-ahead bit from the previous state and that no look-ahead bit is necessary for encoding because the zero is the last bit in data row 10.

You use this procedure to create state machines for the other table rows as well. **Fig Ac**, for example, contains the state machine for data row 11. Because all the state machines have the same beginning state (state zero), you can combine them to form the complete encoding state machine. Furthermore, the state machines may share other states, as shown in the combination of the two data rows (**Fig Ad**).

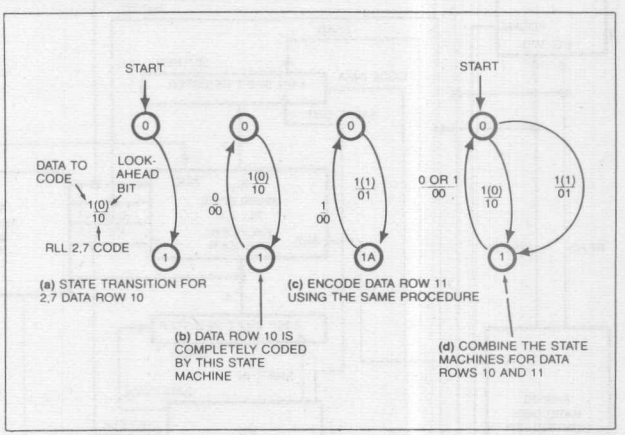


***Fig A—Creating a state machine** from a code table is a 2-part process. First, you create a simple state machine for each row in the table, and then you combine the state machines into one state machine for the whole table.*

the Load signal is high at the same time that Rd_Clk is high, the bits are loaded into the shift register. Because the frequency of Shift_Out is twice that of Rd_Clk, Shift_Out shifts both encoded bits out before the next encoded bits are loaded. Shift_In presents the next data bit to the encoder at the same time that the second encoded bit is shifted out, starting the next encode cycle.

Before the disk-drive system can decode data on the disk, it must synchronize itself with the data clock from the disk and find out when the data bits begin. To assist the circuitry in synchronization and initialization, you can place synchronization signals, as well as a marker indicating the beginning of data, at the beginning of the RLL code. When the RLL circuitry reads these patterns, it's ready to decode RLL data.

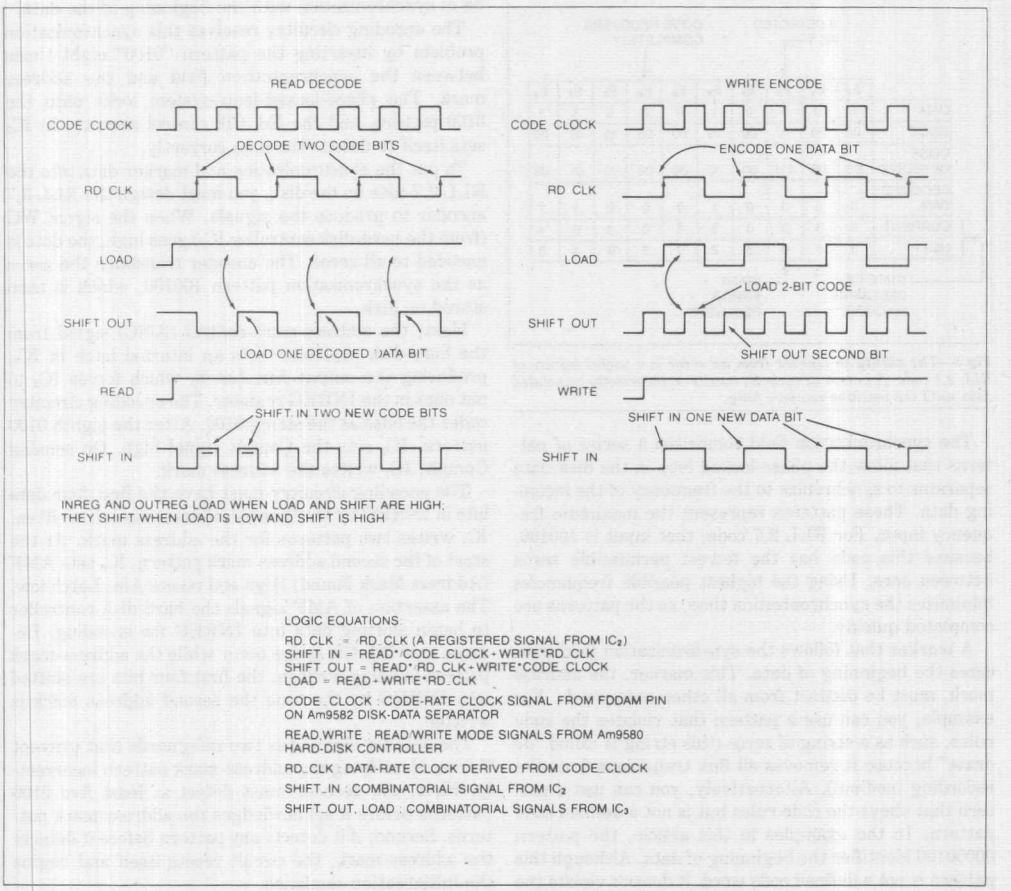
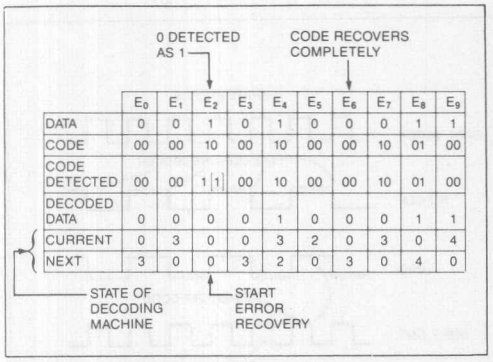


Fig 5—The timing signals from IC_2 and IC_3 control *the loading and shifting of code and data in the INREG and OUTREG registers. Note that the clock and shifting signals for the coded bits have twice the frequency of those for the data bits, a situation that corresponds to the 2:1 density ratio between code and data.*

If the decoder detects any pattern before it detects the address marker, the circuit resets itself and begins the initialization sequence anew.



	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9
DATA	0	0	1	0	1	0	0	0	1	1
CODE	00	00	10	00	10	00	00	10	01	00
CODE DETECTED	00	00	1	1	00	10	00	00	10	01	00
DECODED DATA	0	0	0	0	1	0	0	0	1	1
CURRENT	0	3	0	0	3	2	0	3	0	4
NEXT	3	0	0	3	2	0	3	0	4	0

***Fig 6—The ability to recover from an error** is a useful feature of RLL 2,7 code. An error at cycle E_2 results in incorrectly translated data until the machine recovers fully.*

The synchronization field comprises a series of patterns that allow the phase-locked loop in the disk-data separator to synchronize to the frequency of the incoming data. These patterns represent the maximum frequency input. For RLL 2,7 code, that input is 100100, because this code has the fewest permissible zeros between ones. Using the highest possible frequencies minimizes the synchronization time, so the patterns are completed quickly.

A marker that follows the synchronization field indicates the beginning of data. This marker, the address mark, must be distinct from all other code words. For example, you can use a pattern that violates the code rules, such as a string of zeros (this string is called "dc erase" because it removes all flux transitions from the recording medium). Alternatively, you can use a pattern that obeys the code rules but is not a defined code pattern. In the examples in this article, the pattern 00000100 identifies the beginning of data. Although this pattern is not a defined code word, it doesn't violate the RLL 2,7 code rules as long as it's preceded by no more than two zeros in a row.

When the 100100 pattern is synchronizing the circuitry, the circuitry produces the Rd_Clk signal, which is in phase with the pattern. In order for the decoder to operate properly, the Rd_Clk signal must rise with the first one in the pattern and fall with the second one. However, because of the repetitive nature of the pattern, the circuitry could produce a falling edge of Rd_Clk on the first one in the pattern and a rising edge on the second one, in which case the circuitry would not be in synchronization with the beginning of the data.

The encoding circuitry resolves this synchronization problem by inserting the pattern "0100" eight times between the synchronization field and the address mark. The phase-locked-loop system locks onto the 0100 pattern, and the Rd_Clk control circuitry in IC_2 sets itself to decode the data correctly.

To put the synchronization and marker data into the RLL 2,7 code on the disk, you must design the RLL 2,7 encoder to produce the signals. When the signal WG (from the hard-disk controller IC_3) goes high, the data is encoded to all zeros. The encoder translates the zeros as the synchronization pattern 100100, which is then stored on disk.

Next, the address-mark-control (AMC) signal from the hard-disk controller sets an internal latch in IC_2, producing the output Am_Latch, which forces IC_3 to put ones in the INREG register. The encoding circuitry codes the ones as the string 0100. After the eighth 0100 pattern, IC_2 sets the Complt signal high. On sensing Complt, IC_1 writes the address mark.

The encoding circuitry must have the first four data bits in INREG by the time the address mark is written. IC_1 writes two patterns for the address mark. At the start of the second address-mark pattern, IC_1 sets AMF (Address Mark Found) High and resets Am_Latch low. The assertion of AMF signals the hard-disk controller to begin shifting data into INREG for encoding. Because four Rd_Clk cycles occur while the address-mark pattern is being written, the first four bits are shifted into INREG by the time the second address mark is written.

The decoder circuit has two safeguards that prevent it from identifying the address-mark pattern incorrectly. First, the decoder must detect at least five 0100 patterns before it acknowledges the address-mark patterns. Second, if it detects any pattern before it detects the address mark, the circuit resets itself and begins the initialization sequence.

IC_2 implements these two safeguards by monitoring the RLL code that IC_1 decodes. When the hard-disk controller asserts the RG and AMC signals, IC_2 sets Complt and Am_Latch low. IC_2 sets Complt high when it detects the fifth consecutive 0100 pattern, enabling IC_1 to detect the address mark. If the input to IC_1 is anything other than a 0100 pattern or an address mark, IC_2 sets Complt low, and the initialization begins anew.

If IC_1 successfully detects the address-mark pattern, it sets AMF high. The assertion of AMF causes IC_2 to set Am_Latch high, thus enabling the output of OUT-

REG to send decoded data to the hard-disk controller. Am_Latch remains high as long as RG is high.

RLL 2,7 encoding also provides for error recovery. Whenever a disk-drive system reads code from a disk, the read/write heads or the transmission cables can cause transmission errors. One of the properties of RLL 2,7 code is that any single-bit error (for example, a coded one detected as a zero) will correct itself after a run of at most 16 correctly detected bits.

In short, whenever a 2-bit pattern doesn't match any of the expected patterns for a particular state of the decoding circuitry, the decoding state machine returns to state zero and generates a zero as a translation for the erroneous code bits. Then the decoder shifts the next two code bits into the INREG register and continues decoding from state zero.

In such cases, the decoding state machine can correctly decode coded data, but it may not recover immediately upon returning to state zero. As **Fig 6** shows, although the code bits that the decoder detects may be valid, the decoded data is incorrect because the error has forced that state machine into the wrong state. In this example, the decoder doesn't recover until 6 code bits later (in cycle E6), when it again falls into the correct state and accurately decodes the data. **EDN**

Authors' biographies

Arthur Khu is a product planning engineer at Advanced Micro Devices Inc (Sunnyvale, CA). He is involved in the research and development of architectures for advanced programmable-logic devices, and he has developed a general logic compiler for advanced PLDs. Art holds a BS in Math/Computer Science and an MS in Computer Science from Santa Clara University. He lists racquetball and astronomy among his interests.

Rudolph J Sterner, an engineer at Advanced Micro Devices Inc (Sunnyvale, CA), is engaged in the development of disk-drive-related products. Prior to his two years at AMD, Rudy worked at IMI Corp and Sperry Corp. He holds a BSEE from San Jose State University, and in his spare time he enjoys photography.

Mixing Data Paths Expands Options in System Design

Chip designers are creating powerful CPUs and peripherals with 16- and 32-bit parts. Mixing these with 8-bit parts overcomes limitations imposed by established designs, incomplete families, and software incompatibility.

by Mark S. Young and James R. Williamson

Integrating 16- and 32-bit peripherals and CPUs into 8-bit designs, at the simplest level, means separating the control and data paths from new peripherals and the systems. Mixing different data path widths and control protocols, however, makes possible major improvements in function, performance, and cost.

The price/performance curve of VLSI chips, for example, allows designers to obtain more and better functions for the same amount of money every year. Alternately, the functionality of a device can remain constant while the price falls.

Moreover, these new devices with wider data paths can extend the life of older designs. For example, many of the most popular personal computers today use the 8088 microprocessor and, therefore, are constrained to an 8-bit data path. Designers of add-on accessories for these personal computers prefer the newer 16-bit peripherals. These peripherals will let users preserve their software investments, improve performance, and stave off obsolescence.

Mixing different data path widths can also enhance new designs. For example, it is less expensive to use an 8-bit bus in a new design because the memory requirements are generally cheaper. Only half as many dynamic RAMs are necessary for the same number of kilobytes of memory. In addition, an 8-bit bus needs much less control and support logic. Designers can mix smaller data path peripherals with wider data path CPUs. This allows them to introduce systems based on the newer, more powerful 32-bit CPUs even before 32-bit peripherals are available.

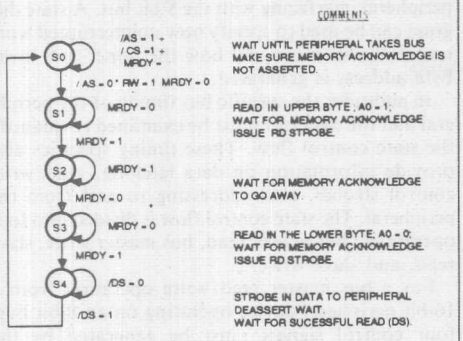
Designers can use this mixing method to obtain wider data paths from existing designs until a new system design is warranted. They can also use parts in unexpected applications. For example, cost-conscious terminal manufacturers might want to use the Am8052/8152A chip set (the 8052 is an advanced CRT controller and the 8152A is a video system controller) in new terminals based on the relatively inexpensive 8051 microprocessor. Mixing the 8-bit, single-chip microprocessor with the 16-bit CRT controller allows designers to maximize the cost/performance ratio of the terminal.

Mixed data path widths can improve bus utilization as well. A 16-bit peripheral in a 32-bit system only occupies half the data bus for data transfers. If the designer mixes the data paths correctly, however, the 16-bit peripheral could transfer data as

Mark S. Young is a product planning engineer at Advanced Micro Devices, Inc (Sunnyvale, Calif). He holds a BA in computer science from the University of California at Berkeley.

James R. Williamson is an applications engineer at AMD. He holds a BS in electrical engineering from the California State Polytechnic University, Pomona.

Reprinted by permission of Computer Design, January 1985.



The state flow control diagram for a bus master read operation illustrates the control sequence employed by the 8/16-bit bus control logic.

32-bit chunks and improve bus efficiency by 100 percent for that peripheral.

Two central concerns stem from mixing devices that communicate over different-sized buses. The first problem results when two devices communicate on a "common" data bus. Consider, for example, a 32-bit system utilizing 8- and 16-bit peripherals. Overcoming the mismatched data paths requires some form of controlled multiplexing/demultiplexing of the different data paths. In addition, extra control signals for partitioning the 32-bit word into 8-, 16-, and 32-bit chunks may be required.

Many 16-bit CPU-based systems that use 8-bit peripherals normally use just the lower 8 bits of the data bus to transfer data to and from the peripheral. This method does not work in systems using 16-bit peripherals and 8-bit CPUs, however, and it tends to break down in systems with 8-bit peripherals having bus master capability.

A bus multiplexing method involves multiple transfers when taking data from or adding data to a mismatched data bus. For example, before a 16-bit peripheral can transfer data over an 8-bit bus, the 16-bit data must be divided into two 8-bit chunks. It is then transferred sequentially. First, the lower 8 bits are transferred out on the bus. Then, in the next transfer cycle, the upper 8 bits of the 16-bit word are sent out. The major difference in the opposite case—a bus read operation from an 8-bit bus to a 16-bit device—is that the first byte read from the system must be latched. Once the second byte has been fetched, the 16-bit peripheral reads in the assembled 16-bit (2-byte) word. Additional provisions may be needed when the 16-bit peripheral only wants to access a single byte.

The other major problem in mixed data path transfers is the actual data read/write operation. The nature of the multiple transfer forces designers to guarantee that the stretched transfer will occur and that it will not be interrupted. Two aspects of stretching the transfer cycle from or to the peripheral illustrate the complexity of this problem.

The first case, when the peripheral is the bus master, is the simplest. A 16-bit peripheral holds its data available for what normally would be two complete bus transfer cycles. This function can be performed when the transfer acknowledge signal to the peripheral is delayed. If the data was latched instead of holding the peripheral in a multiple word transfer, however, the device could try to send the next 16-bit data word and its "new" address. The procedure of latching the data and releasing the peripheral should not be used, therefore, because it may interfere with the addressing of the remaining (pending) 8-bit transfer.

Whenever a device acts as a bus slave to a CPU that cannot access the device's natural word width in a single operation, a different constraint appears. The sequence must be set up so the peripheral cannot obtain the bus while the CPU is in the middle of a slave read/write operation. In a typical system, the CPU is the last device in the interrupt queue. It is possible for the peripheral to become bus master between the first and second read operations and invalidate the results of the first read operation in a realtime system. This is because an 8-bit CPU would have to perform two consecutive read operations to examine a 16-bit peripheral control register.

This function can be handled two different ways. If the CPU has a bus lock instruction, as in the iAPX family of CPUs, the programmer must use one of these instructions before the CPU accesses the peripheral. Alternately, the CPU needs to disable the arbitration logic while it is performing the uninterruptible access with the 16-bit peripheral.

Crucial cycle

The uninterruptible word transfer cycle is crucial for maintaining the integrity of the data transferred. When either the CPU or a peripheral on the bus makes an access using the 8/16-bit control logic, it must complete the larger device's word access before relinquishing the bus. If this requirement is not met, a transfer's integrity can be violated easily by some other device. This interrupts the transfer, and corrupts or aborts the multiplexing sequence.

To illustrate this point, consider a system consisting of an 8-bit CPU and several 8- and 16-bit peripherals. Assume one of the peripherals is executing a block transfer of 16-bit data onto the 8-bit bus. If the CPU interrupted the transfer in order to poll the peripheral during a half-word transfer, two undesirable events would occur. Either the multiplexing

sequence would be damaged irreparably when the CPU polled the peripheral, or the CPU would read garbage from the peripheral.

Designing the control interface to allow mixing of 8- and 16-bit peripherals requires attention to the data and control flow. During a write operation, the data is written out sequentially: the lower byte comes before the upper byte (or vice versa). The read operation differs only because the data bus is 8 bits and because it forgets the last byte transferred; it knows the current byte only. Hence, the interface requires that one of the bytes be latched until the full 16-bit word has been assembled.

The slave mode of operation works almost the same as the peripheral bus master mode. The single exception is the slave write operation. When the interface is defined, the designer must make a conscious choice about which byte (upper or lower) to latch during peripheral read operations (or conversely, slave peripheral write operations). Once this decision has been made, the CPU must always access the latched data byte first (during a slave write) and then access the non-latched byte to complete the transfer. This restriction is minor, requiring no extra software overhead. It could affect the ease of the programmer's coding if not handled properly, however. For example, if the programmer used a compiler to generate the software for the system, extra care may be necessary to ensure the compiler generates the correct addressing sequence.

An alternative solution would be to latch both the upper and lower data bytes. In this case, however, the cost of the interface would increase, as would the complexity, with no appreciable gain. The control flow in these designs derives from two different sources: the state control flow itself and the 16-bit peripheral interfacing with the 8-bit bus. A state diagram can be used to specify how uninterrupted word transfers will occur and how the upper and lower byte address is generated.

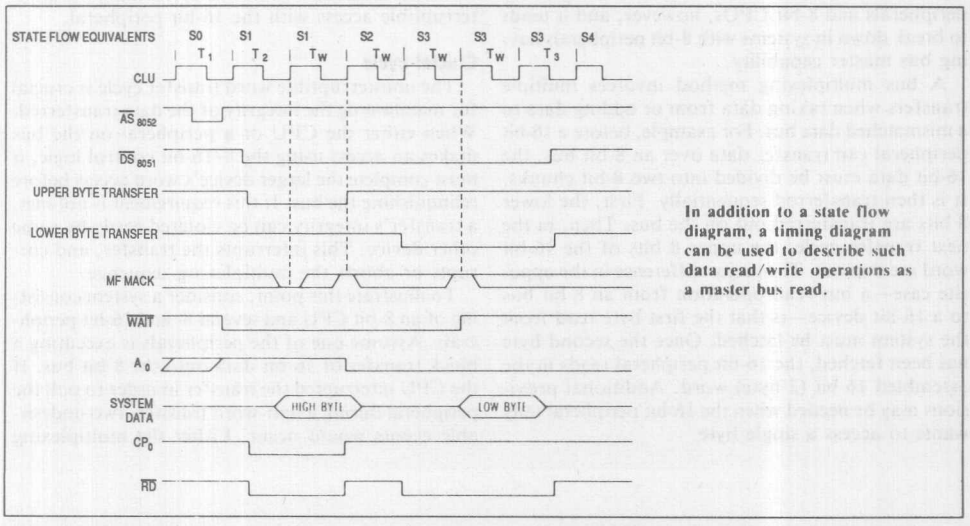
In addition, the specific bus timing of the peripheral and the data bus must be examined to quantify the state control flow. These timing specifics also provide information on data latching, read/write control strobes, and addressing to and from the peripheral. The state control flow is divided into four operations: bus master read, bus master write, slave read, and slave write.

For a bus master read/write operation from a 16-bit peripheral device operating on an 8-bit bus, four control signals must be generated by the 8/16-bit control unit: address bit 0 (A0), peripheral hold ($\overline{WAIT}$), bus read ($\overline{RD}$), and bus write ($\overline{WR}$). The A0 line is generated by the 8/16-bit control logic to indicate which byte is to be transferred in bus master modes only. Otherwise, the A0 generated by the system is used to indicate which byte is being accessed. The $\overline{WAIT}$ line holds up the peripheral during transfers. The $\overline{RD}$ and $\overline{WR}$ lines are required to indicate successive transfer cycles on the bus.

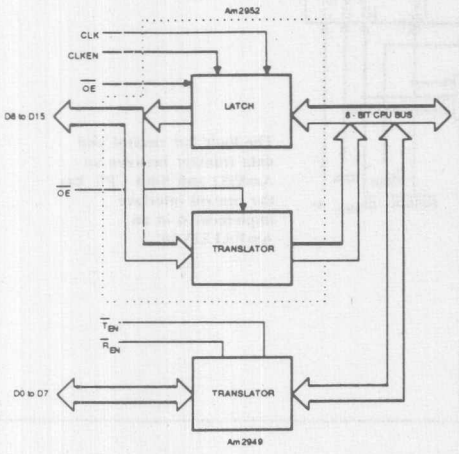
Hidden transfers

The peripheral's signals will only strobe active once because it does not know that two transfers are being executed. The slave transfer flows are almost identical, except the CPU is generating the bus signals and the transfer directions are reversed (ie, a bus write goes into the peripheral).

For this 16- to 8-bit data flow example, the data on the upper byte only needs to be latched when data



In addition to a state flow diagram, a timing diagram can be used to describe such data read/write operations as a master bus read.



The 16-to 32-bit conversation logic diagram indicates the complexity of the bus and tunnel logic control. it must convert between different signal conventions and polarities as well as generate extra functions and bus arbitration control signals.

is being read (as bus master) or written (as a bus slave). An interface to handle this operation needs to latch data coming from the 8-bit data bus into the peripheral, it also needs to act as transceiver when the peripheral is sending data out to the system. A device with a clocked, tri-state output that has an 8-bit wide latch in one direction and a tri-state transceiver in the other direction would be ideal for accomplishing such an interface.

The Am2952 8-bit bidirectional I/O port provides a good enough match to the logic and allows the upper data bus latch and upper data transceiver chips to be combined on one IC. It provides two 8-bit clocked I/O ports, each with tri-state output controls and individual clocks and latch enables. An Am2949 bidirectional bus transceiver completes the logic required for the data path function.

The state flow control requires logic that can move sequentially from state to state, hold in a particular state, and be reset or initialized back to a predefined state. Depending on the number of states required (generally less than 16 distinct states for a design of this complexity), a 3- or 4-bit counter should be able to solve the problem nicely.

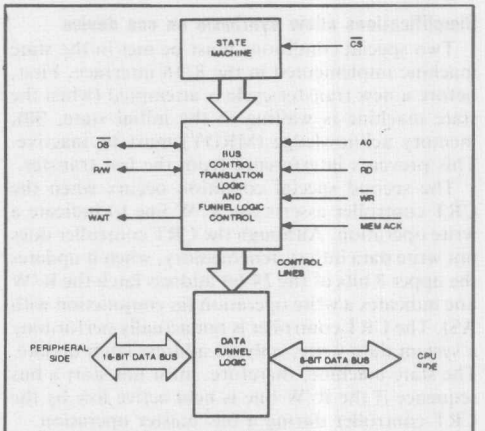
Considerable bus control logic is required to generate the data path flow logic and the bus control signals. This is especially true if the peripherals and CPUs use different signal conventions (eg, when AS, $\overline{DS}$, and $\overline{R/W}$ use address latch enable, $\overline{RD}$, and $\overline{WR}$). Conversion from one signal convention to another, changes in signal polarity, and provision for extra functions (such as generating A0) require a lot of logic synthesis ability. If the peripheral has bus master capability, such additional information as bus arbitration controls must be fed into the next state determination logic in order to decide what control sequence to follow.

Customized interface minimizes cost

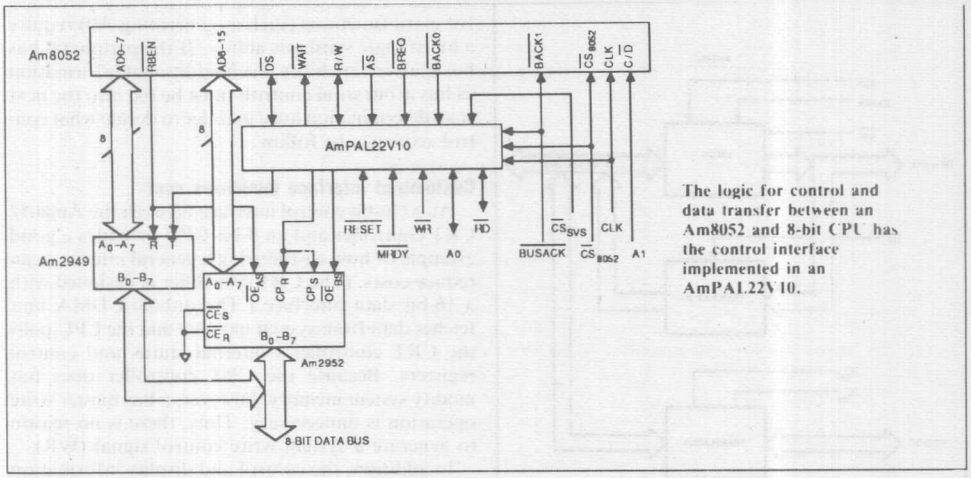
An 8/16-bit control interface between the Am8052 CRT controller and an 8-bit CPU provides a good example of how customizing a general interface can reduce costs. (The CRT controller is designed with a 16-bit data interface.) The onboard DMA unit fetches data from system memory and the CPU polls the CRT controller's internal status and control registers. Because the CRT controller does not modify system memory, however, a bus master write operation is unnecessary. Thus, there is no reason to generate a system write control signal (WR).

In addition, the control and display information must be aligned on word boundaries. This requirement relieves the 8/16-bit control logic from funneling the bytes and performing odd/even byte transfers. It also saves control inputs from the CRT controller because all transfers are words; that is, no need exists for upper and lower data strobes or byte high enable inputs.

The bus master read operations are standard 16-bit data transfers divided into two 8-bit transfers. The CPU's slave accesses are either pointer writes (to select the desired control/status register) or 16-bit data read/write operations. (Pointer write operations



The state machine and the bus and latch controls have to be coupled in order to transfer data between the 8- and 16-bit buses. This generalized machine is designed with the assumption that the peripheral has bus master capability. If this is not the case, the design can be greatly simplified.



The logic for control and data transfer between an Am8052 and 8-bit CPU has the control interface implemented in an AmPAL22V10.

are actually 8-bit operations because only the lower 8 bits of the data form the register address.) The bus master read operation can be represented by a state flow diagram or a timing diagram. Conceptually, state flow diagrams are easier to understand, but timing diagrams usually convey more information. Other state flow diagrams can be derived directly from the timing diagrams of the CRT controller to 8-bit interface.

Simplifications allow synthesis on one device

Two special conditions must be met in the state machine implemented in the 8/16 interface. First, before a new transfer cycle is attempted (when the state machine is waiting in the initial state, S0), memory acknowledge (MRDY) must be inactive. This prevents interference from the last transfer.

The second special condition occurs when the CRT controller asserts the R/W line to indicate a write operation. Although the CRT controller does not write data into system memory, when it updates the upper 8 bits of the 24-bit address latch the R/W line indicates a write operation (in conjunction with AS). The CRT controller is not actually performing a system data write, only an address latch update. The state machine, therefore, must not start a bus sequence if the R/W line is held active low by the CRT controller during a bus master operation.

These simplifications in design allow the CRT controller to 8-bit CPU control interface to be synthesized in a single AmPAL22V10 programmable logic array device. In addition, the bus control signals are converted from AS, DS, and R/W to RD and WR. The minimum CRT controller and bus control signals that must be generated are RD, A0, DS, and R/W. Although the CRT controller uses DS and R/W as inputs during a bus master operation, the PAL device must convert the CPU RD and WR signals to DS and t/W for slave I/O operations.

The signals A0 and RD are generated by the control logic when the CRT controller is performing a read access to system. The WAIT (or not READY) signal to the CRT controller must also be generated by the control logic. The data flow controls require six additional controls to load and strobe the latch, and to enable transceivers to pass data to and from the 8-bit bus. Theoretically, 4 more bits (outputs) are required to represent all the control states needed to manipulate the 8/16-bit control logic. This means the design appears to need 14 output logic units in a PAL device to perform the required task.

Reducing the 14 output cells to the 10 cells available in the PAL device requires a closer look at the timing and output switching functions. The A0 and RD control lines are in effect part of the system bus control and, therefore, cannot be multiplexed easily. The DS and R/W lines to the CRT controller are also fixed because they must be valid throughout the entire transfer cycle as well.

This leaves 6 of the 10 output logic cells of the PAL device to represent the remaining 10 identified control lines. This method of minimization involves careful state synthesis, analysis of the signal switching functions during the transfers, and utilization of several control pins on the CRT controller. By using the BREQ, BACK1, BACK0, CS, and C/D inputs to the PAL device, we can reduce the number of unique states required to 8 instead of 15. This reduces the number of logic cells required for the state machine from 4 to 3 bits.

At this stage, the design requires seven control signals to manipulate the data transfer registers and WAIT line. The two latch enables ($\overline{CE}_S$ and $\overline{CD}_R$) on the Am2952 bidirectional I/O port can be

permanently enabled. By controlling the clock signal to the latches, the controls required for three pins can be reduced to one. The interface control state machine will only use the correct side of the dual latches on the bidirectional I/O port.

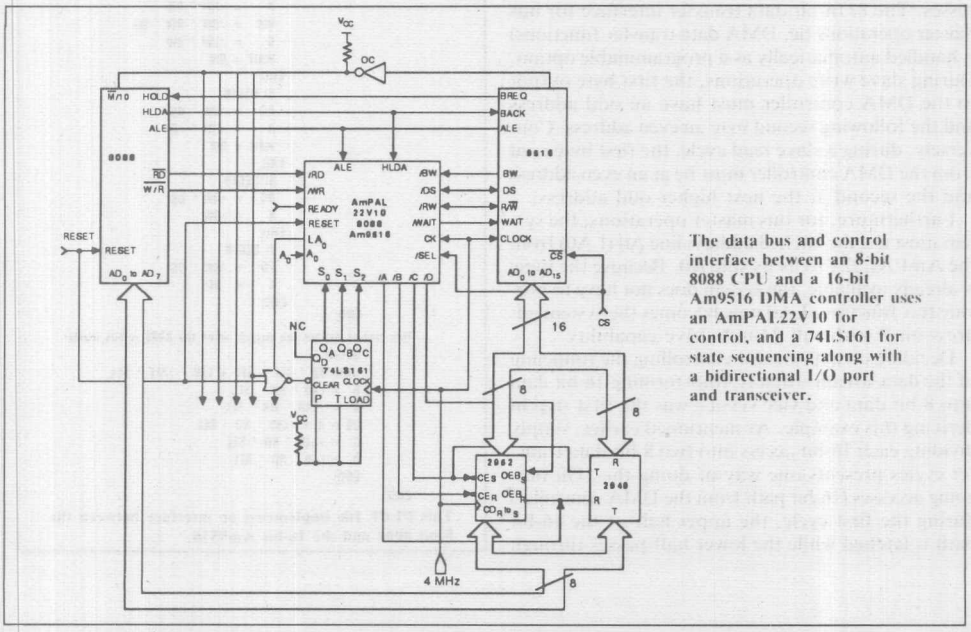
The Am8052 CRT controller helps considerably with its own control bus interface. Two signals provided by the CRT controller, TBEN and RBEN, switch the data transceivers in the correct direction regardless of the type of data transfer (as a bus master or bus slave). When the controller is a bus master performing a read operation, or when it is a bus slave undergoing a write operation, therefore, the RBEN signal is strobed to obtain the correct polarity. By using this line, two of the remaining six control lines can be eliminated (REN on the Am2949 and $\overline{OE}_{AS}$ on the Am2952). Although the $\overline{TBEN}$ line performs a similar function, it does not function correctly in a 16- to 8-bit multiplexed bus environment.

Two of the remaining control lines ($\overline{OE}_{AS}$ on the Am2952 and 10 on the bidirectional bus transceiver) must be generated by individual cells in the PAL device. The two clock enables on the Am2952 are permanently enabled. The two Am2952 clocks are tied together to minimize the amount of logic required in the PAL device used to generate clock strobes to the latches.

This leaves the design with three logic cells and four output functions (the WAIT line to the CRT controller and the 3 state bits). Careful analysis of the state flows and timing diagrams indicates that the WAIT line is only asserted in 4 of the 8 states. A clever assignment of state numbers to the state flow sequence allows the WAIT line to be absorbed into the 3 state encoding bits. The logic equations for the AmPAL22V10 device can be derived directly from the timing diagrams.

An unusual problem might occur when a peripheral device operates as a bus slave on a smaller data bus, such as a 16-bit peripheral to 8-bit CPU. During the first slave write operation, the chip select CS is enabled by the bus master making the access. No actual data—just the data latch—is strobed into the peripheral, however. After the first byte of data has been written, the second access causes the full 16-bit data to be strobed into the peripheral.

If the designer is using a common CS function to both the peripheral and the 8/16-bit control logic, the controller logic must be designed not to glitch or strobe any of the control lines to the peripheral (it must prevent DS, R/W from being enabled, for example). For some peripheral devices, glitches on the control lines might cause the register to be written accidentally onto a register that will be overwritten in the next write cycle anyway. With other peripherals this might be a catastrophic event. Many devices acting as bus slaves have write recovery time requirements (ie, a certain minimum interval between consecutive write operations). Glitches on the control lines might force the next (and final) write operation to be delayed—or cause a violation of the



The data bus and control interface between an 8-bit 8088 CPU and a 16-bit Am9516 DMA controller uses an AmPAL22V10 for control, and a 74LS161 for state sequencing along with a bidirectional I/O port and transceiver.

device specifications. Glitches might evade any special addressing/register accessing scheme used in the peripheral. This might occur, for example, if the slave device requires the user to write the address of the register that was accessed immediately before the register was written. In this case, glitches or useless control strobes could wreck the sequence.

The problem can also be solved by using two lines. In this solution, one of the lines would go to the peripheral device and the other would connect to the 8/16-bit controller. The chip select to the peripheral is activated each time a slave read occurs (for both upper and lower byte accesses), or when a slave write operation occurs and the unlatched 8-bit data is being written. The chip select function to the 8/16-bit controller is chosen each time the peripheral is selected normally (for slave read/writes on both upper and lower 8-bit data transfers). This problem is bypassed completely when two separate chip select functions are used: one for loading up the Am2952 latch during a slave write/read and one to strobe the Am8052 controller into action when it is needed by the 8-bit CPU.

Bus conversion maximizes flexibility

A data bus and control interface to an 8088 8-bit microprocessor and Am9516 16-bit DMA controller can be created using four devices: an AmPAL22V10 for the control block, a 74LS161 counter for the state sequencer, an Am2952 bidirectional I/O port, and an Am2949 bidirectional transceiver.

This design incorporates certain simplifications. The DMA controller requires word accesses only during command chaining and for slave register accesses. The 8/16-bit data transfer interface for bus master operations (ie, DMA data transfer functions) is handled automatically as a programmable option. During slave write operations, the first byte output to the DMA controller must have an odd address and the following second byte an even address. Conversely, during a slave read cycle, the first byte read from the DMA controller must be at an even address and the second at the next higher odd address.

Furthermore, for bus master operations, the system must use the latched address line A0 (LA0) from the AmPAL22V10 as its sole A0. Because the logic is already available, the system does not have to provide this function. LA0 now becomes the system address bit 0 with full 24-mA drive capability.

Deciding on a means for controlling the funneling of the data stream—that is, transforming 16-bit data into 8-bit data and vice versa—was the first step in deriving this example. As mentioned earlier, simply dividing each 16-bit access into two 8-bit data transfer cycles presents one way of doing this. On outgoing accesses (16-bit path from the DMA controller) during the first cycle, the upper half of the 16-bit path is latched while the lower half passes through

```
PIN
        CK      = 1     /RD     = 23
        S[0:2]  = 2:4   /WR     = 22
        A0      = 5     LA0     = 21
        /SEL    = 6     /DS     = 20
        ALE     = 7     /RW     = 19
        HLDA    = 8     /WAIT   = 18
        /BW     = 9     /A      = 17
        READY   = 10    /B      = 16
        RESET   = 11    /C      = 15
                        /D      = 14;
BEGIN
        IF (RESET) THEN ARESET( );
This section defines the wiggles when the Am9516 is bus master
        IF    (HLDA) THEN ENABLE( );
        IF    (/S[2] * HLDA\ THEN BEGIN
              IF (S[1] * /S[0]) THEN
                    LA0   = /CK * BW + /BW * A0 *
                            ALE + / BW * LA0 * /ALE ;
              ELSE
                    LA0   = BW + /BW * A0 *
                            ALE + /BW * LA0 * /ALE  ;
        END;
        IF    (HLDA) THEN
              (CASE) (S[2:0])
                    BEGIN
                     1) BEGIN
                      RD   = /RW * DS                ;
                      A    = /BW * /RW * /CK         ;
                      WR   = /BW * RW * DS           ;
                      C    = /BW * RW                ;
                      WAIT = 1                       ;
                     END;
                     2) BEGIN
                      RD   = /RW * DS                ;
                      B    = BW                      ;
                      A    = /BW * /RW               ;
                      WR   = /BW * RW * DS           ;
                      C    = /BW * RW                ;
                      WAIT = /BW
                     END;
                     3) BEGIN
                      RD   = /RW * DS * B            ;
                      B    = BW * CK                 ;
                      A    = /BW * RD                ;
                      WR   = /BW * RW * DS           ;
                      C    = /BW * RW                ;
                      WAIT = BW                      ;
                     END;
                     5) BEGIN
                      RD   = /RW * DS                ;
                      A    = /BW * /CK               ;
                      WAIT = BW                      ;
                     END;
                     6) BEGIN
                      RD   = /RW * DS                ;
                      A    = /BW                     ;
                     END;
                     7) BEGIN
                      RD   = /RW * DS                ;
                      A    =  RD                     ;
                     END;
        END;
This section defines the wiggels when the 8088 is bus master
        BEGIN
        LA0= A0 * ALE + SEL + LA0 * /ALE * SEL
        B  = LA0 * WR * SEL
        A  = /LA0 * WR * SEL
        DS = A + /LA0 * RD * SEL
        C  = /LA0 * RD * SEL
        D  = LA0 * RD * SEL
        END;
END.
```

This PLPL file implements an interface between the 8-bit 8088 and the 16-bit Am9516.

Programming the PAL and the counter

In writing the Programming Language for Programmable Logic (PLPL) file to control the operation of the AmPAL22V10 and the 74LS161 counter, the inputs to the PAL device from the counter are assigned S0, S1, and S2, respectively. Then, it is possible to apply a "sculptured design" technique to the entire timing diagram (see figure in Panel, "A matter of timing") by using the Case statement from PLPL. By assigning combinatorial equations to only one binary partition or column at a time (Case), the designer can ignore all other aspects of the design for the time being and generate simple equations directly from the timing waveforms.

During clock time T1 of the Am9516's word read cycle the state of the 74LS161 (S0, S1, S2) is cleared to 000 by the assertion of address latch enable (ALE). LA0 is the only output control signal from the CRT controller asserted during this period. This signal is handled as a special case, however. During time T2 of the DMA controller's word read cycle, the $\overline{RD}$ and $\overline{WAIT}$ outputs from the CRT controller must be asserted. This time partition corresponds to the state inputs S2, S1, S0 = 001. Therefore, the first Case equations are

```
CASE     (S[2:0])
  BEGIN
  1)  BEGIN
        RD =  /RW*DS  ; Transform Control
                      ; Signals /RW and DS
                      ; into Intel /RD

        WAIT = 1      ; Assert     Wait
                      ; unconditionally

  END;
```

During time T2 of the DMA controller's byte read cycle, $\overline{A}$ is the only additional output not already accounted for in the Case statement. This signal allows a byte of data to flow through the bidirectional bus transceiver into the DMA controller during byte read operations. Some additional constraints are placed on this signal, however: it must only be asserted in time T2 on byte read operations (the B/W input) and it must be delayed by a half clock period from the rising edge of T2 (CK signal). Thus the Case statement becomes

```
CASE   (S[2:0])
  BEGIN
  1)  BEGIN
        RD =  /RW*DS          ;
        A =  /BW*/RW*/CK      ; enable the
                              ; receiver

        WAIT = 1

  END;
```

Finally, by examining the last time T2 elements ($\overline{WR}$ and C) during the DMA controller's byte write cycle, the remaining terms in Case 1 are derived. With the exception of LA0, the remaining equations were developed in the same fashion. Clearly, this "sculptured" technique is a very simple and methodical means for arriving at the Boolean requirements for a logic block.

As the PLPL listing shows, the signal LA0 was handled slightly differently from the previously discussed method. The number of product terms generated via the Case statement made this approach necessary. The number exceeded the upper limit (16 terms) for a programmable logic array. As a practical matter, therefore, it was necessary to optimize this signal manually. However, it should be noted that this step will not be necessary once the fully optimized version of PLPL becomes available.

a tri-state buffer onto the 8-bit bus. During the second cycle, the tri-state buffer is turned off and the previously latched half of the data is driven onto the bus. On incoming accesses (8-bit path to 16-bit path), the process is reversed.

The control mechanisms that perform this cycling depend on the WAIT and R/W signals passing to and from the DMA controller, and on the ability to enable or disable the latches and transceivers selectively. The Am2952 bidirectional I/O port was chosen because of its dual registers and its flexible control. The AmPAL22V10 device was chosen to match the required number of control pins and functions. Since the complexity of this design requires the use of all of the PAL's I/O pins for control functions, however, it was necessary to use a 74LS161 counter to provide the state sequencer function.

Programming with PLPL

It has long been the logic designer's "art" to merge the often very different concepts and notations of timing information with Boolean logic. Yet, the evolution of a syntax to fully express this art has taken a long time. AMD recently developed such a language for programming the AmPAL22V10, however.

"Programming Language for Programmable Logic," or PLPL, allows the designer to specify a design using multiple input formats. This specification flexibility supports the variety of design approaches necessary to express different design problems efficiently. These formats range from simple sum-of-products Boolean equations to high level constructs. PLPL also supports the input specifications for many types of AND/OR based devices, including all of the current AMD programmable logic array and PROM devices.

PLPL is block structured, and includes the high level language constructs If-Then-Else, Case, and For; all familiar to many programmers of the C and Pascal languages. Macros, functions, constants, and variables may also be used in PLPL. The language also facilitates use, clarity, and self-documentation.

Such current programmable logic technology and associated programming languages as PLPL allow

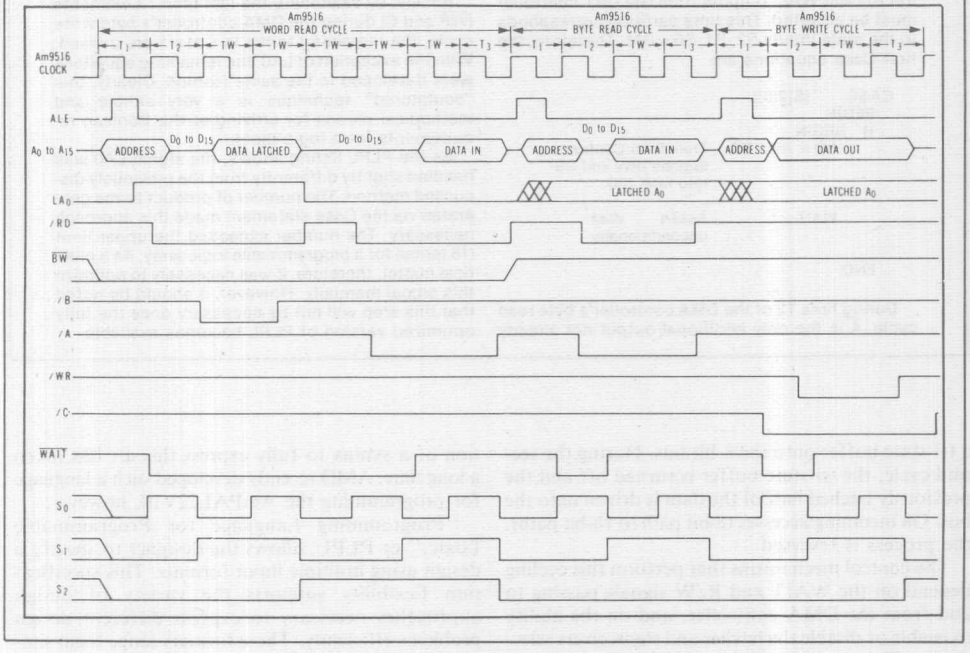
A matter of timing

The complex AmPAL22V10 design used the accompanying timing diagram to correspond to the desired waveforms. They are partitioned by the respective binary state (or count) from the counter.

The desired timing requirements during the period when the DMA controller is bus master appears below. During time T1, address latch enable (ALE) is asserted by the DMA controller to denote the beginning of the cycle; a short time later, an address is driven onto the bus. This address is valid at the falling edge of ALE. The control signal LA0 (latched A0), therefore, must be valid at this time, as well. In this phase of the cycle, it must also be high to enable the odd byte from memory to be loaded into the bidirectional I/O port. In addition, the assertion of ALE performs the function of resetting the 74LS161 counter to 0000 in order to synchronize the cycle.

During time T2, the DMA controller will assert its $\overline{DS}$ signal. The timing for this signal, in conjunction with the R/$\overline{W}$ signal (asserted in T1) must be transformed into an 8088-equivalent $\overline{RD}$ signal. During a word read cycle, this $\overline{RD}$ signal also must be artificially negated and then reasserted to accomplish a double byte read. At the same time, the DMA controller must be "parked" in order to multiplex or assemble a word. Thus, the $\overline{WAIT}$ signal is also asserted at time T2. During time TW (S2, S1, S0 = 010), the receiver clock enable control signal $\overline{B}$ must be asserted in order to allow the next system clock's rising edge to strobe the upper byte into the bidirectional I/O port. This is accomplished during the next TW period (S2, S1, S0 = 011).

During the remainder of the word read cycle, $\overline{RD}$ is negated and then reasserted after LA0 has been forced low to address the even byte. $\overline{A}$ is then asserted to allow both the previously latched upper byte and the current lower byte to be driven onto the DMA controller's pins. And finally, the $\overline{WAIT}$ signal is negated, allowing the DMA controller to finish its read cycle by strobing in the 16 bits of command data on its data pins.



highly organized application-oriented control blocks to be formed easily. These tools can conceptually raise the designer above the details of the design at the logic level and directly translate the necessary response characteristics from a timing diagram.

This approach can be referred to as a "sculptured design" technique because it is analogous to the way solid stone is formed according to an artist's image. Raw logic can be transformed directly into useful control functions from the desired timing information.

The AmPAL22V10 is, in essence, a fuse-programmable gate containing up to 22 inputs and 10 outputs. It can define and program that architecture of each output on a pin by pin basis. Thus, the designer is free to optimize the design mix between registered and combinatorial functions as needed.

The AmPAL22V10 is programmed by opening fusable links in any or all of its 10 output macrocells, as well as in its AND gate array. The AND gate structure is very similar to other PAL devices; therefore

it allows the same powerful, yet familiar features. However, it is the AmPAL22V10's 10 output logic macrocells that give the designer substantial new design freedom. Moreover, at each macrocell output is a tri-state output buffer controlled by a separate output-enable AND gate.

These macrocells provide the AmPAL22V10's key features. They can be configured to make any or all of the I/O pins act either in sequence or in combination and have either active-high or active-low characteristics. Furthermore, the output enables can individually control the direction of the pins so they act as outputs, inputs, or bidirectional ports.

A number of trade-offs and limitations are apparent in a design that so dramatically affects the input and output of the system. The most obvious limitation stems from under utilization of 16-bit peripherals on an 8-bit bus—the speed of all I/O operations are cut in half. As a result, bus utilization will increase if the 16-bit peripheral represents a significant factor of the bus use. A CRT controller such as the Am8052 might use 5 to 10 percent of the bus bandwidth for display information when using 16-bit I/O. Converting to 8-bit I/O would double bus use to 10 to 20 percent. Another factor that might affect the bus usage is the efficiency of the 8- to 16-bit conversion control logic. If the state machine designed to perform the 8/16-bit (or 16/32-bit) conversion is improperly designed, extra transfer overhead might be introduced. This might mean a sequential transfer of two 8-bit values would take twice as long a single 16-bit transfer.

The design constraints might limit the use of the peripheral to byte-only operations during data transfers (as in the design using the DMA Am9516 controller), and slow it down by a factor of two during command operations. For such a DMA device as the Am9516, the extra time required for command fetching is not usually a significant portion of bus time.

System designers will have to weigh the cost of the extra overhead on a case-by-case basis. The benfits may well justify these limitations—particularly when the bus is self-limiting, but the device characteristics allow for value-added designs. In addition to bus degradation for certain configurations, extra logic and design effort are involved. Most interfaces outside a system's immediate family require some kind of extra interface logic, however. By manipulating the signals and incorporating them into programmable logic devices such as the AmPAL22V10 device, therefore, most of this logic is free.

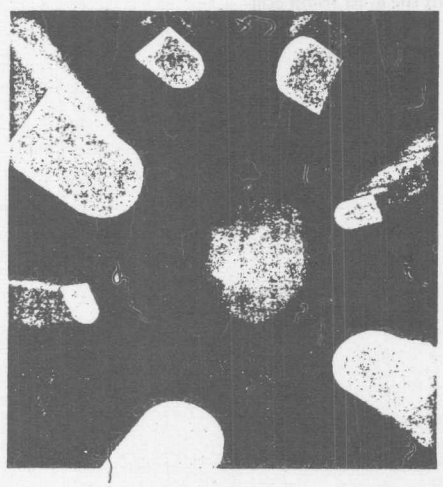
Programmable Logic Chip Rivals Gate Arrays in Flexibility

Hanging a user-customized macrocell at each output of an LSI-level fuse-programmable array logic gives it the flexibility of a gate array. The result: Less hardware is needed for high-level functions.

Reprint with permission from ELECTRONIC DESIGN

Designers of semicustom ICs have long appreciated the in-house programming capabilities of programmable array logic devices. But gate arrays, with their LSI densities and wider opportunities for customization, are an inviting alternative, even if they must be sent out for metal-mask processing. Which approach should the designer use?

An agonizing choice between convenience and versatility is no longer necessary. The first fuse-programmable chip capable of implementing LSI circuits blends the advantages of both types. The result is more cost-effective designs with higher-level functions than previous programmable logic devices offered and a shorter turn-around time than is possible with gate arrays.



The new chip, the AmPAL22V10, can be thought of as a fuse-programmable gate array, since at twice the density of previous programmable array logic devices, it replaces the equivalent of logic circuits having 500 to 1000 gates (see "Blending Programmable Logic and Gate Arrays," p. 97). In fact, because of its flexible, programmable architecture, the chip spans the breadth of standard SSI, MSI, and present-generation PAL devices, often accomplishing what could not be practically done with several such chips.

Designers program the 22V10 by opening fusable links in any or all of its 10 output macrocells, as well as in its AND-gate array, developing the needed logic functions. Though the AND-gate structure makes the unit similar to other PAL devices, the chip's 10 output logic macrocells afford a substantial new degree of design freedom (Fig. 1).

What's more, at the chip's output pins are 10 three-state output buffers, each fed by a macrocell and controlled through a separate output-enable AND gate.

The macrocells provide the 22V10's key features: They can be configured to make any or all of the I/O pins act either sequentially or combinatorially and have either active highs or active lows. Additionally the Output Enable can individually control the I/O terminals to act as output pins, input pins, or bidirectional ports. Thus the designer can readily change the chip's architecture.

Within the array, each AND gate is fed by the true and complementary levels of the chip's 12 input lines, one of which doubles as a clock line. Also, each AND gate is fed by the true and complementary levels of the 10 feedback outputs from the output

Brad Kitson, Section Manager
David Laws, Managing Director
Warren Miller, Section Manager
Advanced Micro Devices Inc.
901 Thompson Place, PO Box 3453
Sunnyvale, Calif. 94088

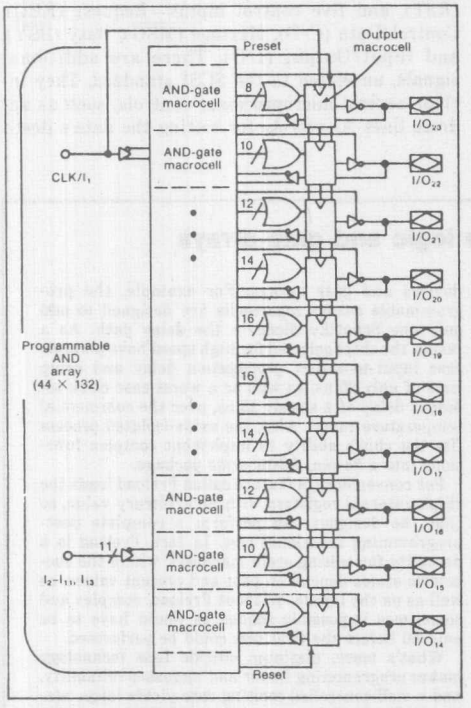
Semicustom ICs: Programmable Logic

logic macrocells.

Each of the AND gates therefore has 44 inputs, and there are a total of 132 AND gates in the array. The result is over 5800 fuses that can be programmed to perform the desired function.

A total of 120 of the input AND gates are distributed to drive 10 OR gates, forming the logical part of the AND array. In addition, one AND gate is assigned to control each of the 10 three-state outputs, and the remaining two gates activate synchronous register presets and asynchronous register resets.

The array itself consists of 10 AND-gate macrocells (five similar pairs), each logically ORing the outputs of anywhere from 8 to 16 AND gates. The AND gates are distributed so that the two outermost macrocells in the array logically OR together eight AND gates. The next two outermost macrocells logically OR together 10 AND gates; the next, 12 AND gates; and so on, traveling inward along the array. In other words, moving toward the array's center, two AND gates are added to successive pairs of macrocells, ending with the two innermost macrocells, each ORing 16 AND gates apiece.



1. A fuse-programmable LSI device combines the efficiency of programmable logic arrays with the flexibility of gate arrays. Ten user-definable output macrocells allow the designer to set the logic outputs individually for one of four configurations. They can even be used as inputs or dynamically controlled I/O.

The variable distribution of AND gates is particularly useful when implementing counting, exclusive-OR functions, and complex state machines. For example, a 10-bit binary counter has only one AND gate on the LSB but requires an additional gate for each succeeding bit, for a total of 10 AND gates to implement the MSB. Should a hold, parallel load, or specialized function also be desired, the MSB requires even more AND gates.

To program the array into its familiar sum-of-products Boolean function, a designer blows the corresponding fuse links. Those links initially connected the chip's pins and their complements to all 132 of the 44-input AND gates. As mentioned, the designer can also program any OR-gate output to be a dedicated input or a dynamically controlled I/O pin. This is possible because the connection from an output macrocell to its output pin is through a three-state output buffer and is controlled by an output-enable AND gate. Turning off the connection turns the output pin into an input. Otherwise the pin remains connected to the output macrocell, fed by the OR gate.

An output with a difference

The chip's output section is where it appears more like a gate array than a PAL device. The output macrocells allow the designer, by individually setting the outputs to be either sequentially or combinatorially and either active high or active low, to program either the true or complementary version of a logical term—whichever makes the most efficient use of the chip's AND array and fits the chip to the application. Furthermore, independent control over the active state helps the designer conform to common logic conventions, thus avoiding confusion and error.

For example, certain functions, such as chip selects and resets, tend to be active low. The output macrocells avoid contradictions with convention by inverting outputs, individually changing them when necessary to conform. Finally, by individually selecting either sequential or combinatorial outputs, the macrocells impart to the chip the capacity for implementing more functions than is possible with existing PALs.

The macrocell (Fig. 2a) contains three main logic blocks: a rising-edge-triggered D-type flip-flop with asynchronous reset and synchronous preset inputs, a four-to-one output-path selection multiplexer, and a two-to-one feedback-path selection multi-

control the four possible output configurations: R_n selects sequential or combinatorial operation; P_n selects active low or active high.

With both fuses intact, the output is sequential and active low (Fig. 2b). The Q output of the flip-flop passes to an inverting output buffer, and the Q output feeds back to the AND array internally. This configuration is particularly suitable for state-machine designs. Alternatively, by programming (blowing) P_n, the designer can make the output active high.

On the other hand, by programming only R_n, the designer makes the output combinatorial and active low. For this scheme, the OR-gate output bypasses the flip-flop and feeds the inverting output buffer directly. At the same time, the feedback multiplexer passes the output back into the array (Fig. 2c).

By programming both fuses, the designer makes the output combinatorial and active high, and, again, the feedback multiplexer passes the output back to the array. This configuration serves best for "glue" logic and for generating complex control functions.

Finally, when the device is programmed in the combinatorial configuration, the programmable Output Enable can be used to transform an output pin into a dedicated input or a dynamically controlled I/O terminal. Once enabled as a dedicated input, however, a pin's output function is sacrificed. This capability is extremely valuable when partitioning functions into the device. The limitation is no longer the numbers of input devices, but the total number of device pins. Lastly, as a dynamically controlled I/O, the pin serves both functions, meeting most bus requirements.

Intelligent use

One of the most challenging jobs in designing microprocessor systems is installing intelligent peripheral controllers—that is, controllers that can transfer data themselves and so unburden the processor from inefficient polling operations. One solution is to build a custom intelligent controller from scratch using the 22V10 (Fig. 3). This controller governs data flowing between a microprocessor and up to eight peripheral devices, all conforming to the Small Computer Systems Interface standard (SCSI, the ANSI X3T9.2/82-2).

The interface consists of an 8-bit bidirectional data port (DB_7-DB_0); three control outputs—Acknowledge ($\overline{ACK}$), Select ($\overline{SEL}$), and Reset ($\overline{RST}$); and five control inputs—Request ($\overline{REQ}$), Control/Data ($\overline{C}/D$), Message ($\overline{MSG}$), Busy ($\overline{BSY}$), and Input/Output ($\overline{I}/O$). There are additional signals, unrelated to the SCSI standard. They include several microprocessor controls, such as address lines A_1 and A_2 for setting the data's desti-

Blending programmable logic and gate arrays

Centered on a fuse-programmable AND array, the architecture of a PAL (programmable array logic) device offers designers a powerful, instantly modifiable logic building block. As a result, PAL devices are used in a wide variety of applications, ranging from microprocessor-based systems to superminicomputers.

Since gate arrays are not programmed in house, they have a much slower turnaround time and higher cost. Yet gate arrays' simple logic structure makes them easier to customize than PAL devices, and their larger size makes for denser logic structures.

A gate array's simpler structure can be a drawback, however. For example, typical gate arrays are based on just two- or three-input gates; therefore, to implement complex functions, they often require many gate levels, which results in slow logic structures. In contrast, PAL devices, like memories, make more efficient use of their silicon area and can be configured into fast logic structures. Consequently, even with a slight overhead for programming circuitry, PAL densities are beginning to rival those of gate arrays.

However, innovative circuit techniques and an oxide-isolated bipolar process team up in the AmPAL22V10 and combine the advantages of PAL devices and gate arrays. For example, the programmable output macrocells are designed to add only one Schottky diode in the delay path. As a result, the chips selected for high speed have a worst-case input-to-output propagation delay and setup time of only 25 ns, as well as a worst-case clock-to-output delay of a speedy 15 ns, over the commercial temperature range. Also, the oxide-isolated process fits the chip's ability to implement complex functions into a 24-pin, 0.3-in.-wide package.

For convenience, a feature called Preload loads the chip's internal registers with an arbitrary value, so that the designer can perform a complete post-programming functional test. In fact, Preload is a necessity for testing state machines, where the successive states depend on past and present values, as well as on the inputs. Without Preload, complex and sometimes impossible sequences would have to be entered before the first test could be performed.

What's more, platinum silicide fuse technology makes programming faster and increases reliability, and a well-controlled melting rate yields large, stable, nonconductive gaps. Address circuitry, much like that of a PROM, blows one fuse at a time, and programming is done with an algorithm that is easily adapted to a wide variety of available machines.

Semicustom ICs: Programmable Logic

nation, read and write controls ($\overline{IORD}$ and $\overline{IOWR}$), Interrupt Request ($\overline{INTREQ}$), Chip Select ($\overline{CSSCSI}$), System Clock ($\overline{SYSCLK}$), and buffer and latch enables ($\overline{BE}$ and $\overline{LE}$) for passing bus data between the processor and its peripherals.

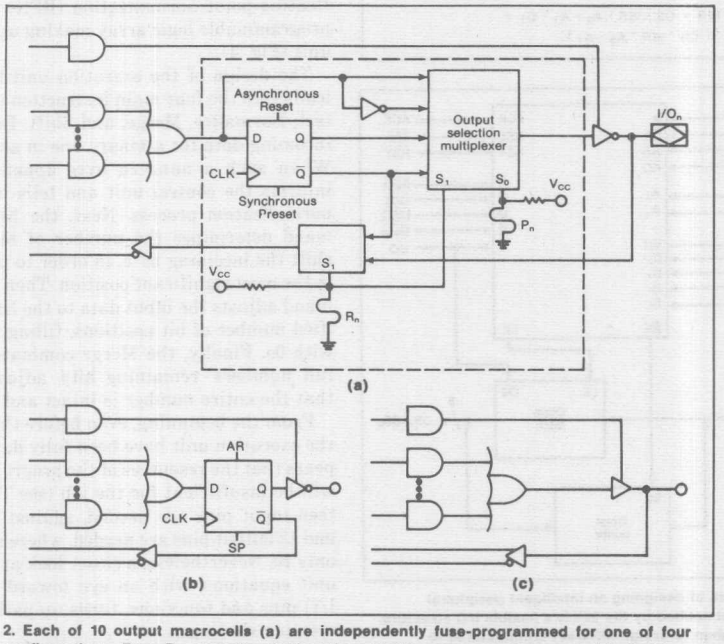
In operation, the controller assigns one of eight peripheral devices to be a master by activating $\overline{SEL}$ and at the same time sends the master's address to the SCSI data port. From then on, when a device notifies the controller that it needs to transfer either a command or data, the controller interrupts the host processor. In turn, the processor reads the controller's status register to determine which device is requesting service and the type of request it is making. After its status register is read, the controller removes the interrupt request and carries out the required transfer.

Meanwhile the selected device issues a $\overline{BSY}$ signal, letting the other devices know that the bus is in use. The data transfer is managed by the handshake signals $\overline{ACK}$ and $\overline{REQ}$, and the processor activates $\overline{RST}$ as needed by pulling both $\overline{WR}$ and $\overline{RD}$ low at the same time.

Matching the chip to its application starts with an assessment of the available resources against those needed to do the job. In general, that means establishing that the chip has enough input and output pins and product terms to accommodate the logic functions required by the application.

As to the pinouts, the 10 outputs required by the application do, in fact, match the 10 available on the chip. Four of the pins are used for the bidirectional data bus and the rest are dedicated outputs. As for inputs, the SCSI controller requires 11, which the chip more than satisfies with its total of 12.

Next the designer develops the product terms needed to execute the controller's job and compares the number and size of the required terms with the number available from the chip. The resulting controller equations (see Table 1) number 10, and these nestle neatly into the chip's 10 available AND-gate macrocells. Also, the maximum number of terms in any one equation—in this case, those making up the acknowledgment function—is only 6, compared with the chip's capacity of 8 to 16. Thus the controller function fits easily into a single programmable



2. Each of 10 output macrocells (a) are independently fuse-programmed for one of four configurations. Fuse R_n controls whether the outputs are sequential or combinatorial, and fuse P_n determines whether they are active high or active low. With no fuses blown, the output is sequential and active low (b). Blowing only R_n gives a combinatorial active-low output, and blowing only P_n gives a sequential active-high output. Finally, blowing both fuses changes the output to combinatorial and active high (c).

Semicustom ICs: Programmable Logic

logic array, as can more complicated microprocessor peripheral interfaces.

The 22V10's real power, however, becomes apparent when it is needed to fill an application that appears to demand more than the chip's resources can deliver. Insufficient input or output pins or product terms are frequent shortcomings of many devices when the job is very formidable. With its highly versatile I/O architecture, however, the 22V10 can accommodate conditions that would make chips even larger than itself inadequate.

Table 1. Peripheral controller equations

Combinatorial equations

$BE = CS \cdot RD \cdot A_2 \cdot A_1$

$LE = CS \cdot WR \cdot A_2$

$D_3 = A_1 \cdot C + A_1 \cdot ACK$

$D_2 = A_1 \cdot M + A_1 \cdot SEL$

$D_1 = A_1 \cdot B + A_1 \cdot RST$

$D_0 = A_1 \cdot I + A_1 \cdot R$

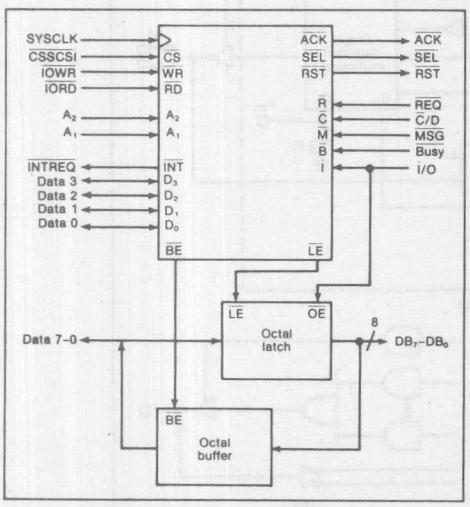
Sequential equations

$INTREQ = R \cdot (I \cdot C \cdot M + J \cdot C \cdot M + I \cdot C \cdot M + ICM)$

$ACK := CS \cdot WR \cdot A_2 \cdot A_1 \cdot D_3 + R \cdot ACK \cdot (\overline{CS \cdot WR \cdot A_2 \cdot A_1}) + R \cdot CS \cdot RD \cdot A_2 \cdot A_1$

$SEL = CS \cdot WR \cdot A_2 \cdot A_1 \cdot D_2 + B \cdot SEL \cdot (\overline{CS \cdot WR \cdot A_2 \cdot A_1})$

$RST = RD \cdot WR + CS \cdot WR \cdot A_2 \cdot A_1 \cdot D_1 + RST \cdot (\overline{CS \cdot WR \cdot A_2 \cdot A_1})$



3. The problem of designing an intelligent peripheral controller is simplified by the 22V10's flexible I/O structure. With the freedom to assign different architectures to different pins, the designer can make better use of the chip's input and output pins and product terms.

For example, one of the most complicated functions to perform with a digital computer is floating-point arithmetic, or more specifically the normalizing of numbers in the process. As a result, no standard MSI or LSI logic for normalization exists. Yet the programmable array is well-suited to integrating at least part of this complicated logic function.

Normalization is necessary because a floating-point number is represented in two parts: a mantissa and an exponent. The mantissa is a fractional number that is adjusted, or normalized, so that a binary 1 always occupies the most significant bit. To normalize a floating-point number without changing its value, a compensating adjustment must be made to the exponent. In effect, the exponent, a power of two, is incremented for each bit position that the mantissa is shifted to the left.

Because the process of normalization can be very costly if done wholly in parallel, it is more efficient to process a number one byte at a time. Furthermore, because of the function's complexity, it is reasonable to partition the normalization unit into two units: an execution unit, which acts on the incoming data, and a control unit, which directs the execution unit. Thus the two units together form a byte-wide floating-point normalization (BFN) unit, with the programmable logic array making up the execution unit (Fig. 4).

The design of the execution unit breaks up logically into the four main instruction categories; Detect, Normalize, Merge, and Shift. Detect tests the incoming data for a binary one in any bit position. When such a nonzero byte appears, a flag, F, informs the control unit and tells it to begin the normalization process. Next, the Normalize command determines the number of bit positions to shift the incoming byte, in order to put its first bit in the most significant position. Then the Shift command adjusts the input data to the left by the specified number of bit positions, filling the lower bits with 0s. Finally, the Merge command adds in the full numbers' remaining bits, adjusting them so that the entire number is intact and shifted.

From the beginning, even before the equations of the execution unit have been fully developed, it appears that the resources of the programmable array will be insufficient for the job (see Table 2). Fourteen input pins are needed, against the chip's 12, and 12 output pins are needed, whereas the chip has only 10. Nevertheless, a closer look at the execution unit equations, with an eye toward consolidating I/O pins and functions, turns up paths around the seeming shortages.

One "problem" that proves not to be one is the number of outputs. Although they number 13, not

all are active at the same time. For example, the outputs V_0-V_2, which specify the number of shift positions as a three-bit binary number, are active only when no other outputs are. Thus they can share their output circuitry with other signals, relieving the output shortage.

However, combining the shift outputs with other terms creates a product-term deficiency. Yet, again, careful observation reveals a solution. One way to reduce the product terms of a function is to invert it, using the programmable array's output macrocell. The most likely candidate for such a reduction is the function with the most product terms. Although there are several, the best, because it is relatively removed functionally from the others, is the detection flag, F. Its function is given by:

$$F = D_7 + D_6 + D_5 + D_4 + D_3 + D_2 + D_1 + D_0$$

where D_7 through D_0 are input data bits, which, if any are high, indicate a nonzero byte.

When De Morgan's theorem is applied, the long OR string of active-high data reduces to a single AND term, saving a total of seven product terms. Still, two input pins must be found for the programmable array to fully incorporate the logic of the execution unit. Since De Morgan's theorem cannot minimize the logic any further, the only answer is to locate any product terms that underuse the chip's resources, combine them, and lend their output pins to the cause by making them I/O pins.

Examination of the instruction input specification (see Table 3) reveals that during Detect only three of the five instruction inputs are necessary, freeing the two others to be used as outputs. Therefore instruction bits I_0 and I_1 can share the outputs that were previously dedicated to shift bits V_0 and V_1, thus meeting the execution unit's final requirements in a single chip.□

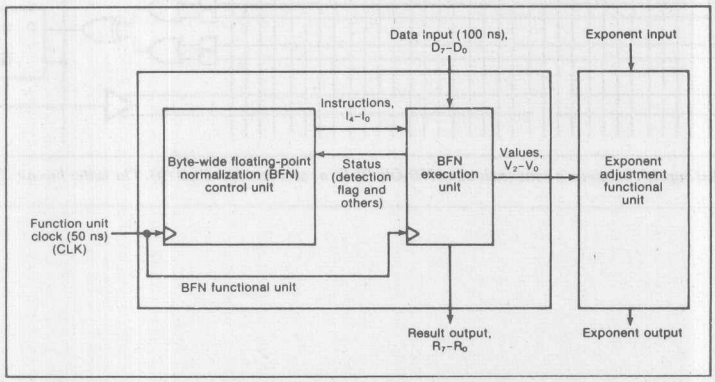
2

Table 2. Allocation of chip resources

	Input pins		Output pins		
	Number	Names	Number	Names	Product terms
Initial requirements	14	CLK, D_7-D_0, I_4-I_0	12	F (detection flag), V_2-V_0, R_7-R_0	114 required
Resources	12	None	10	None	120 available
Final requirements	12	CLK, D_7-D_0, I_4-I_2, I_1/V_1, I_0/V_0	12	I_1/V_1, I_0/V_0, R_7/V_2, R_0/F, R_6-R_1	108 required

Table 3. Instruction definitions

	Instruction code (I_4-I_0)	Detection flag
Detect	00XXX	0
Normalize	00XXX	1
Merge	$01N_2N_2N_0$	X
Shift	$10N_2N_1N_0$	X



4. Floating-point arithmetic is a hardware-intensive process that, for the most part, has defied integration. Yet by partitioning the tasks into smaller pieces, a designer can apply the power of the programmable array to the job. The chip's flexibility contributes to its successful implementation as a floating-point execution unit, even though its resources may seem insufficient.

XOR PLDs Simplify Design of Counters and Other Devices

XOR PLDs' unique architecture is better suited than that of conventional AND-OR PLDs to certain applications. Once you understand the basic theory behind their operation, you can easily configure them to perform the tasks of such devices as binary counters and video shift registers.

Chris Jay, *Monolithic Memories Inc*

Many designs—notably high-speed counters for accessing memories, for counting events, or for sequential control in high-speed processor systems—require state machines that step through their sequence of operation by a binary count. XOR PLDs are ideal devices for these types of jobs, but they are rarely considered for such applications—perhaps because design engineers don't understand their useful, but unusual, architecture.

Like the more sophisticated of today's conventional devices (**Fig 1a**), exclusive-OR (XOR) versions have a registered architecture (**Fig 1b**). The distinguishing features of the latter are two pairs of product terms (multiple-input AND gates), which drive two summing terms (OR gates), which in turn feed two inputs of an XOR gate. The combinational XOR gate's output connects directly to the D input of the register. (For a theoretical discussion of the advantages of XOR PLDs over AND-OR PLDs, see **box,** "Principles of XOR PLD operation.")

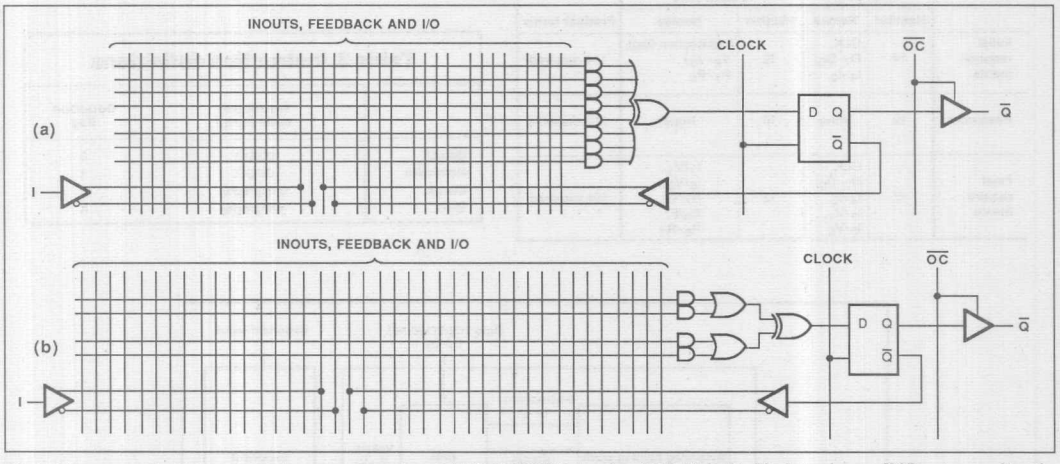


Fig 1—You can easily distinguish between a conventional AND-OR PLD (a) and an XOR PLD (b). The latter has an XOR gate preceding the register.

Reprinted from EDN, May 28, 1987
© 1987 CAHNERS PUBLISHING COMPANY
a Division of Reed Publishing USA.

The distinguishing features of an XOR PLD are two pairs of product terms, which drive two summing terms, which in turn feed two inputs of an XOR gate.

The applications described here make use of MMI's 20X8A and 20X10A PAL devices. These 24-pin XOR PLDs have an improved 22.2-MHz clock and worst-case propagation delay of 30 nsec compared with older, 12.5-MHz, 50-nsec parts.

XOR PLDs bring benefits to counters

The prime use of XOR PLDs is in counters. **Fig 2a** shows the pinouts and **Fig 2b** the Palasm logic equations of a 10-bit, loadable, up/down counter. You can design such a counter into a single 20X10A PAL device. The counter has ten registers, Q0 through Q9, capable of binary, up/down counting. The up control is high for an up count and low for a down count. Registers Q1 through Q8 load synchronously through data inputs D1 through D8; the loading is enabled by an active-low /LD input (see **box,** "Palasm notational conventions"). Because of a pin limitation of the package, Q9 loads via the up/down-control pin. This pin (the D9 input during loading) controls the counter's up- or down-counting mode. Q0 is always loaded with a logic low.

A counter larger than 10 bits requires two or more XOR PLDs. Unlike MSI counters, which come with carry-in and carry-out pins, you must incorporate in your design provisions for cascading XOR PLDs. When cascading two XOR PLDs, you must provide a carry-out-enable signal from the least significant counter to the count-enable input of the most significant counter.

When the least significant counter has reached its maximum count, it should enable the most significant counter to increment its count after the next rising edge of the clock pulse. However, time delays arising from the time needed to decode and propagate the Carry signal could slow the system down. To avoid this delay, your design should generate a look-ahead-carry output.

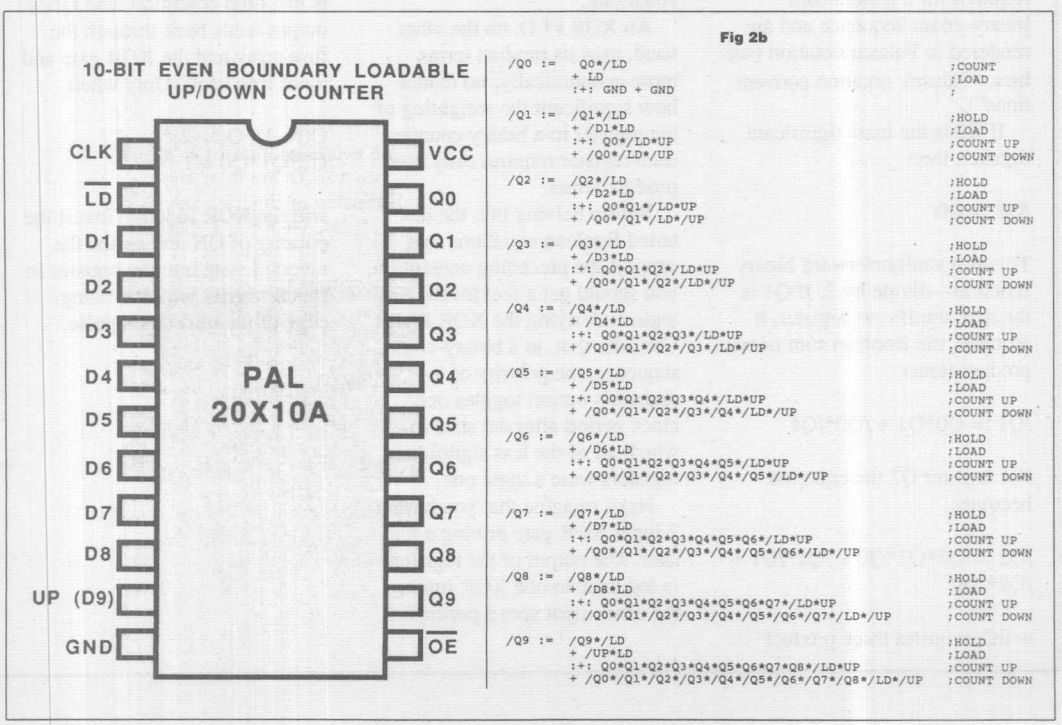


Fig 2b

```
/Q0 :=   Q0*/LD                                          ;COUNT
       + LD                                              ;LOAD
       :+: GND + GND                                     ;

/Q1 :=  /Q1*/LD                                          ;HOLD
       + /D1*LD                                          ;LOAD
       :+: Q0*/LD*UP                                     ;COUNT UP
       + /Q0*/LD*/UP                                     ;COUNT DOWN

/Q2 :=  /Q2*/LD                                          ;HOLD
       + /D2*LD                                          ;LOAD
       :+: Q0*Q1*/LD*UP                                  ;COUNT UP
       + /Q0*/Q1*/LD*/UP                                 ;COUNT DOWN

/Q3 :=  /Q3*/LD                                          ;HOLD
       + /D3*LD                                          ;LOAD
       :+: Q0*Q1*Q2*/LD*UP                               ;COUNT UP
       + /Q0*/Q1*/Q2*/LD*/UP                             ;COUNT DOWN

/Q4 :=  /Q4*/LD                                          ;HOLD
       + /D4*LD                                          ;LOAD
       :+: Q0*Q1*Q2*Q3*/LD*UP                            ;COUNT UP
       + /Q0*/Q1*/Q2*/Q3*/LD*/UP                         ;COUNT DOWN

/Q5 :=  /Q5*/LD                                          ;HOLD
       + /D5*LD                                          ;LOAD
       :+: Q0*Q1*Q2*Q3*Q4*/LD*UP                         ;COUNT UP
       + /Q0*/Q1*/Q2*/Q3*/Q4*/LD*/UP                     ;COUNT DOWN

/Q6 :=  /Q6*/LD                                          ;HOLD
       + /D6*LD                                          ;LOAD
       :+: Q0*Q1*Q2*Q3*Q4*Q5*/LD*UP                      ;COUNT UP
       + /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/LD*/UP                 ;COUNT DOWN

/Q7 :=  /Q7*/LD                                          ;HOLD
       + /D7*LD                                          ;LOAD
       :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*/LD*UP                   ;COUNT UP
       + /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/LD*/UP             ;COUNT DOWN

/Q8 :=  /Q8*/LD                                          ;HOLD
       + /D8*LD                                          ;LOAD
       :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*Q7*/LD*UP                ;COUNT UP
       + /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/Q7*/LD*/UP         ;COUNT DOWN

/Q9 :=  /Q9*/LD                                          ;HOLD
       + /UP*LD                                          ;LOAD
       :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*Q7*Q8*/LD*UP             ;COUNT UP
       + /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/Q7*/Q8*/LD*/UP     ;COUNT DOWN
```

***Fig 2a—This diagram shows the pin designations** for a 20X10A XOR PLD configured as a 10-bit, loadable, up/down counter. This design is not cascadable.*

Principles of XOR PLD operation

The configuration of an exclusive-OR (XOR) PLD simplifies counters and other state machines that progress through a binary-encoded counting sequence. You do not, of course, necessarily need a PLD with XOR gates to design a binary counter; you can use the simpler, sum-of-products (SOP) architecture of a conventional PLD (see **Fig 1a** in the main text) to implement the logic equations that define a binary-count sequence.

However, the conventional PLD requires a larger number and AND gates, or product terms, for a counter's higher-order registers than does an XOR PLD. The Boolean equations that follow prove this point. They are required for a generalized binary-count sequence and are rendered in Palasm notation (see **box**, "Palasm, notation conventions").

If Q0 is the least significant register, then

/Q0 := Q0

This is a straightforward binary function—divide by 2. If Q1 is the nest significant register, it requires, the Boolean sum of two product terms:

/Q1 := Q0*Q1 + /Q0*/Q1

For register Q2 the equation becomes

/Q2 ;= Q2*Q1*Q0 + /Q2*/Q1 + /Q2*/Q0

which requires three product terms. From a generalized counter with N registers, the Nth register, QN, requires N+1 product terms:

/QN := QN*Q(N–1)* … Q1*Q0
+ /QN*/Q(N–1)
…
+ /QN*/QN1
+ /QN*/Q0

For example, the most significant register is a 10-bit counter, Q9, would require the summing of 10 product terms, using 10 AND gates, in a convential AND-OR PLD. Consequently, although you can write abstract counter equations for AND-OR PLDs, you may not be able to find a real PLD with enough product terms to embody those equations.

An XOR PLD, on the other hand, uses its product terms more economically; no matter how significant the weighting of the register in a binary counter, each register requires only two product terms.

Before delving into the detailed Boolean equations that support the preceding contention, you should get a feel for the logic underlying the XOR PLD. Consider that, in a binary-count sequence, the polarity of a register's output toggles one clock period after the state in which all of the less significant registers were a logic one.

Next, imagine that you have a 2-input XOR gate driving a register. The output of the register is fed back to one XOR input; the other input sees a product term comprising all the less-significant digits. The XOR gate will hold the register in its existing state until the product terms of the less-significant digits go True, whereupon the XOR gate will toggle the output of the register.

The following is a generalized equation for a register (QN) in a binary-count sequence using the XOR PLD:

QN := QN :+: Q(N–1)*Q(N–2)*
… Q2*Q1*Q0

You can consider the term Q(N–1)*Q(N–2)*… Q2*Q1*Q0 as a Carry input to register QN.

When Q(N–1)*Q(N–2)*… Q2*Q1*Q0 = 0, the register QN is in a hold condition, and QN's output feeds back through the fuse array and the XOR gate and loads into itself. Only when

Q(N–1)*Q(N–2)*…
Q2*Q1*Q0 = 1

will the XOR function invert the polarity of QN and strobe the inverted state into the register in synchronicity with the rising edge of the next clock pulse.

Unlike MSI counters, which come with carry-in and carry-out pins, you must incorporate in your design provisions for cascading XOR PLDs.

The logic in the least significant counter can decode the penultimate count and feed the D input of a carry-output-enable register. When the counter increments to the final count, the carry-output-enable signal is then clocked into the carry-output-enable register. The propagation of the carry-output-enable signal to the next counter stage is therefore coincident with the generation of the output of the ultimate count from the

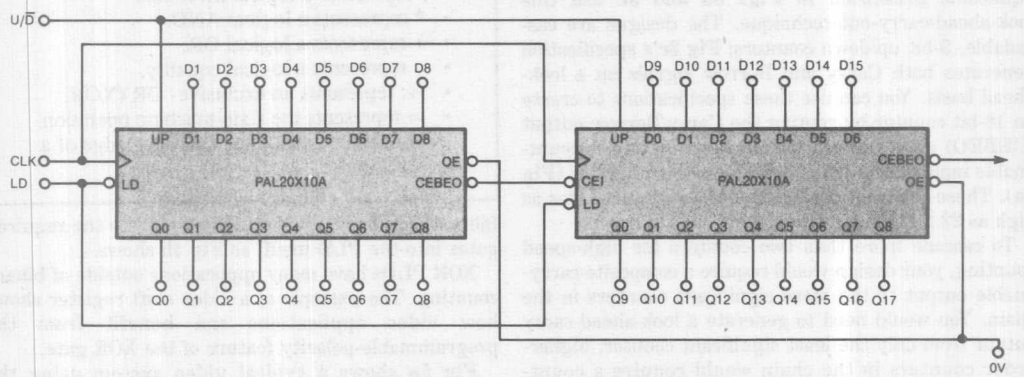


Fig 3a—Unlike the counter illustrated in Fig 2a, *this counter uses a look-ahead-carry-out technique. Provision of this signal renders the counter cascadable.*

Fig 3b

```
CEBEO   :=  UP*/Q0*Q1*Q2*Q3*Q4*Q5*Q6*Q7*Q8*/LD
        +   /UP*Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/Q7*/Q8*/LD

/Q0     := Q0*/LD
        +  LD

/Q1     :=  /Q1*/LD
        +   /D1*LD

        :+: Q0*/LD*UP
        +   /Q0*/LD*/UP

/Q2     :=  /Q2*/LD
        +   /D2*LD
        :+: Q0*Q1*/LD*UP
        +   /Q0*/Q1*/LD*/UP

/Q3     :=  /Q3*/LD
        +   /D3*LD
        :+: Q0*Q1*Q2*/LD*UP
        +   /Q0*/Q1*/Q2*/LD*/UP

/Q4     :=  /Q4*/LD
        +   /D4*LD
        :+: Q0*Q1*Q2*Q3*/LD*UP
        +   /Q0*/Q1*/Q2*/Q3*/LD*/UP

/Q5     :=  /Q5*/LD
        +   /D5*LD
        :+: Q0*Q1*Q2*Q3*Q4*/LD*UP
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/LD*/UP

/Q6     :=  /Q6*/LD
        +   /D6*LD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*/LD*UP
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/LD*/UP

/Q7     :=  /Q7*/LD
        +   /D7*LD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*/LD*UP
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/LD*/UP

/Q8     :=  /Q8*/LD
        +   /D8*LD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*Q7*/LD*UP
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/Q7*/LD*/UP
```

Fig 3c

```
CEBEO   := UP*Q0*Q1*Q2*Q3*Q4*Q5*Q6*Q7*Q8*/LD*CEI
        + /UP*/Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/Q7
        */Q8*/LD*CEI

/Q0     :=  /Q0*/LD*/CEI
        +   Q0*/LD*CEI
        :+: LD*/D0

/Q1     :=  /Q1*/LD
        +   /D1*LD
        :+: Q0*/LD*UP*CEI
        +   /Q0*/LD*/UP*CEI

/Q2     :=  /Q2*/LD
        +   /D2*LD
        :+: Q0*Q1*/LD*UP*CEI
        +   /Q0*/Q1*/LD*/UP*CEI

/Q3     :=  /Q3*/LD
        +   /D3*LD
        :+: Q0*Q1*Q2*/LD*UP*CEI
        +   /Q0*/Q1*/Q2*/LD*/UP*CEI

/Q4     :=  /Q4*/LD
        +   /D4*LD
        :+: Q0*Q1*Q2*Q3*/LD*UP*CEI
        +   /Q0*/Q1*/Q2*/Q3*/LD*/UP*CEI

/Q5     :=  /Q5*/LD
        +   /D5*LD
        :+: Q0*Q1*Q2*Q3*Q4*/LD*UP*CEI
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/LD*/UP*CEI

/Q6     :=  /Q6*/LD
        +   /D6*LD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*/LD*UP*CEI
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/LD*/UP*CEI

/Q7     :=  /Q7*/LD
        +   LD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*/LD*UP*CEI
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/LD*/UP*CEI

/Q8     :=  /Q8*/LD
        +   LD
        :+: Q0*Q1*Q2*Q3*Q4*Q5*Q6*Q7*/LD*UP*CEI
        +   /Q0*/Q1*/Q2*/Q3*/Q4*/Q5*/Q6*/Q7*/LD*/UP*CEI
```

registers in the least significant counter. By anticipating the ultimate count in this fashion, this scheme avoids the propagation delay involved in decoding the final count.

The counter designs specified by the sets of logic equations presented in **Figs 3b** and **3c** use this look-ahead-carry-out technique. The designs are cascadable, 9-bit up/down counters; **Fig 3c**'s specification generates both Carry and Borrow signals on a look-ahead basis. You can use these specifications to create an 18-bit counter by routing the Carry/Borrow output (CEBEO) of the first up/down counter to the count-enable input (CEI) of the second up/down counter (**Fig 3a**). These counters are capable of clocking at rates as high as 22.2 MHz.

To cascade more than two counters for high-speed counting, your design would require a composite carry-enable output to the more significant counters in the chain. You would need to generate a look-ahead-carry output from only the least significant counter; higher-order counters in the chain would require a count-enable output generated from the ultimate, or final, count rather than the penultimate count. **Fig 4a** shows a technique for cascading more than two counters in a synchronous-count chain. Depending upon the availability of input pins, you could incorporate the required gates into the PLD itself, as **Fig 4b** shows.

Palasm notation conventions

For consistency with the design spercifications, the text of this article uses the Palasm notational conventions for Boolean operators instead of the classical symbols. Palasm is MMI's PAL assembler, which converts Boolean-logic equations into fuse maps for PLDs. *(Ed Note: Readers familiar with other PLD compilers may be more used to the C-language conventions used in Cupl and Abel. One compiler, logIC from Kontron, allows you, thankfully, to choose your own favorite symbol set.)*

- / represents a logical inversion.
- * represents a logical AND.
- + represents a logical OR.
- = represents a logical equality.
- :+: represents an exclusive- OR (XOR.
- := represents the state-machine operation "Becomes equal after the rising edge of a clock pulse."

XOR PLDs have many applications outside of binary counting. The example of a video shift register shows how video applications can benefit from the programmable-polarity feature of the XOR gate.

Fig 5a shows a typical video system using the PAL20X8A. **Fig 5b** gives the PLD design specification. An XOR PLD, when functioning as a video-shift register, loads row data, in parallel, from a character ROM or RAM. A control input to the XOR PLD (reverse

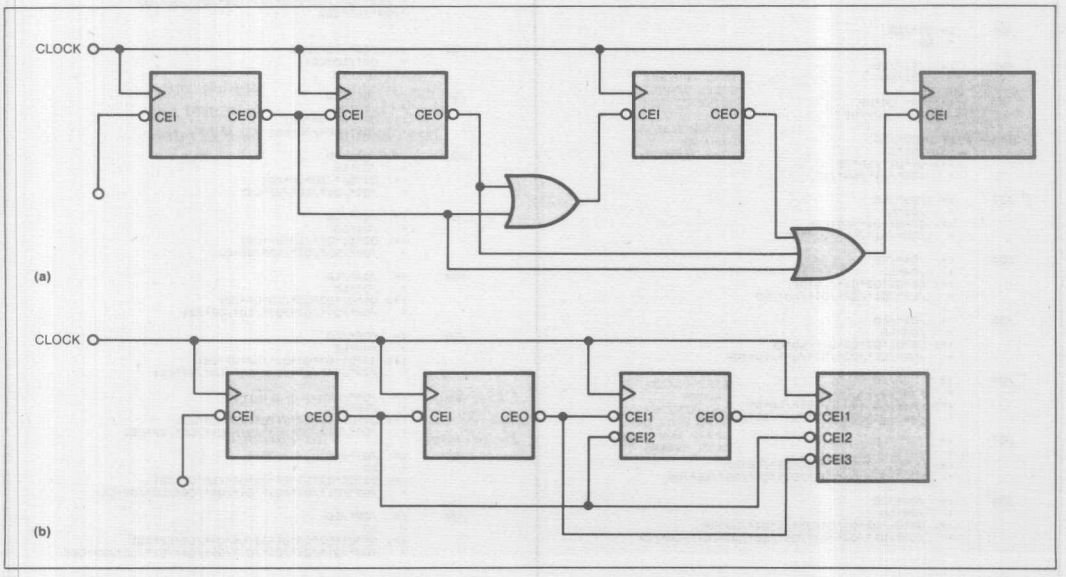


*Fig 4—**You can cascade two or more XOR-PLD counters** in a synchronous-count chain with either external **(a)** or internal **(b)** logic.*

The XOR PLD series has many applications outside of binary counting.

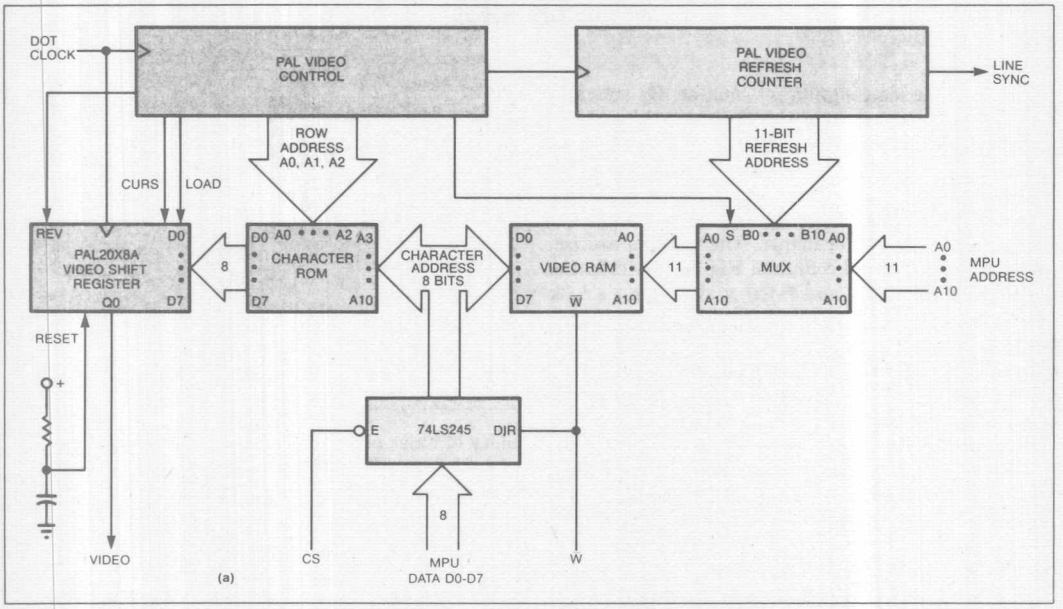


Fig 5a—XOR PLDs suit other applications besides counters. *The device's programmable output polarity is a vital part of this design, which uses an XOR PLD as a video shift register that incorporates a cursor control.*

Fig 5b

```
/Q0 := /D0*LD*/REV*/CURS*/RST          ;LOAD D0
     + D0*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D0
     :+: /Q1*/LD*/CURS*/RST            ;SHIFT FROM Q1
     + RST                             ;RESET Q0 LOW
                                       ;
                                       ;CURSOR LOW CLEARS /Q0
/Q1 := /D1*LD*/REV*/CURS*/RST          ;LOAD D1
     + D1*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D1
     :+: /Q2*/LD*/CURS*/RST            ;SHIFT FROM Q2
     + RST                             ;CURSOR LOW CLEARS /Q1
                                       ;RESET CLEARS Q1
/Q2 := /D2*LD*/REV*/CURS*/RST          ;LOAD D2
     + D2*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D2
     :+: /Q3*/LD*/CURS*/RST            ;SHIFT FROM Q3
     + RST                             ;CURSOR LOW CLEARS /Q2
                                       ;RESET CLEARS Q2
/Q3 := /D3*LD*/REV*/CURS*/RST          ;LOAD D3
     + D3*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D3
     :+: /Q4*/LD*/CURS*/RST            ;SHIFT FROM Q4
     + RST                             ;RESET CLEARS Q3
                                       ;CURSOR LOW CLEARS /Q3
/Q4 := /D4*LD*/REV*/CURS*/RST          ;LOAD D4
     + D4*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D4
     :+: /Q5*/LD*/CURS*/RST            ;SHIFT FROM Q5
     + RST                             ;CURSOR LOW CLEARS /Q4
                                       ;RESET CLEARS Q4
/Q5 := /D5*LD*/REV*/CURS*/RST          ;LOAD D5
     + D5*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D5
     :+: /Q6*/LD*/CURS*/RST            ;SHIFT FROM Q6
     + RST                             ;CURSOR LOW CLEARS /Q5
                                       ;RESET CLEARS Q5
/Q6 := /D6*LD*/REV*/CURS*/RST          ;LOAD D6
     + D6*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D6
     :+: /Q7*/LD*/CURS*/RST            ;SHIFT FROM Q7
     + RST                             ;CURSOR LOW CLEARS /Q6
                                       ;RESET CLEARS Q6
/Q7 := /D7*LD*/REV*/CURS*/RST          ;LOAD D7
     + D7*LD*REV*/CURS*/RST            ;LOAD INVERSE OF D7
     :+: /LD*/CURS*/RST                ;SHIFT HIGH INTO /Q7
     + RST                             ;RESET CLEARS Q7
                                       ;
```

video, or REV) determines whether or not the data loaded into the register is to be inverted. Therefore, the XOR PLD can supply normal-video or reverse-video information.

This design also incorporates a cursor control, which, when active, sets all of the register's outputs high. The design also has a reset line to clear all the registers. When you release the load input, the clock input will serially clock the data out of register Q0 as a video-data stream. The clock input to the PLD determines the dot rate of the video information and must come from an external dot-rate generator.

The Palasm logic-equation files presented in this article omit comments, pin definitions, and device-simulator specifications because of space limitations. If you want the entire Palasm file, you may call the author at (408) 970-9700. **EDN**

Author's biography

Chris Jay is applications manager at Monolithic Memories Inc (Santa Clara, CA), where he has worked for four years. He previously worked for Fairchild and Texas Instruments in the UK. Chris obtained a BSc in electronics from Essex University in England, and he is a member of the Institute of Electrical Engineers, London. His hobbies include photography, electronics, and traveling.

IMPORTANT
NOTES

The PAL20RA10 Story—The Customization of a Standard Product

AR-157

Among the greatest challenges one faces when designing with the new generation of 32-bit, high-performance microprocessors is maintaining both byte and word addressability within the microprocessor's 32-bit-wide word format and accommodating both high- and moderate-speed memory components and peripherals in the same system. Consider this example: Motorola's 32-bit microprocessor, the MC68020, has a feature known as dynamic bus sizing. This feature helps maintain both byte and word addressability within the microprocessor's 32-bit word format. The microprocessor has internal logic that manipulates and directs words smaller than 32 bits. However, the MC68020 requires additional external logic to handle 8- and 16-bit memory arrays and peripherals and to maintain byte addressability of 32-bit memory arrays. Consider this as well: The MC68020 is available in both 12-MHz and 16.7-MHz speed grades. The 16.7-MHz speed translates into 60-ns machine cycles and, consequently, into 90-ns memory cycles. Thus, the MC68020 can keep up with many of today's fastest memories. Moreover, in systems that use both fast memories and slower, low-cost bulk ones, the 16.7-MHz version can interface with the slower memories without holding back the faster ones. But to do this, the MC68020 must insert wait states to extend the memory cycle for the slower memories.

Both design problems call for additional circuitry—that is, some "glue" logic. And because the MC68020 employs asynchronous control signals and the memories need synchronous control, the additional circuitry must be capable of both asynchronous and synchronous operation. A product from Monolithic Memories, Inc., the PAL20RA10, provides what we believe is an elegant, single-chip solution to these problems.

A new, registered, asynchronous PAL device originated with suggestions from the manufacturer's customers.

The PAL20RA10 is one of the most recently developed members of Monolithic's programmable array logic product family. As with all other PAL devices, it contains an array of programmable links that the user can configure to implement a variety of logic functions. The PAL20RA10 is unusual in that the customer not only can configure the internal logic transfer function but also can implement logic functions to control the clock, the output enables, and the set and reset operations for each register. In addition, the user can configure the outputs to be registered or nonregistered, and active high or active low. We will discuss these features, as well as a universal port interface adapter application, in greater detail later in this article.

PLDs and the semicustom market

Programmable logic devices, or PLDs, fit between standard logic and customer-designed circuits and thus are considered semicustom circuits, along with gate arrays and standard cells. PAL products are a type of PLD. Discussions of semicustom integrated circuits often center on technological issues: process capabilities, device speed and power dissipation, number of available gates, and so on. But the customization of a device involves more than just meeting the customer's functional requirements. The device

Copyright © 1986 The Institute of Electrical and Electronics Engineers, Inc.
Reprinted with permission from IEEE MICRO,
10662 Los Vaqueros Circle, Los Alamitos, CA 90720.

Table 1.
Logic devices rated by various selection criteria.

Criterion	Standard logic	PLDs	Gate arrays	Standard cells	Full custom
Sourcing and availability	1	1	3	4	4
Performance	5	3	3	1	1
Space efficiency	5	4	2	2	1
Ease of design*	2	1	2	4	4
Ease of design changes	3	1	2	4	4
Design cost**	1	1	3	4	5
Time to market	2	1	3	4	5
System-level mfg. cost	5	3	3	2	1

* With the exception of standard logic, this means ease of use of the design tools, i.e., ease of understanding the design tool concepts, ease of doing the required input, etc.
** Design cost is the sum of the NRE (non-recurring engineering) costs and such non-recurring manufacturing costs as masking, test vector generation, etc., that are part of the product development cycle.

should also meet the customer's requirements for price, availability, risk, ease of use, and support. Only when all of these requirements have been met has a device been truly customized.

Within this definition, even standard 7400 series logic might be considered customized. It meets most customers' requirements for low cost, availability, low risk, and high ease of use, and it provides just about any desired function.

PLDs and gate arrays—semicustom alternatives and trade-offs

The PLD was first conceived as a replacement for discrete TTL logic designs. Customers wanted higher integration than that possible with standard SSI/MSI logic, but they could not find the functions they wanted in standard LSI circuits and did not want to pay for fully customized ones. The PLD provided a new option—semicustom LSI—that had the advantages of SSI/MSI availability and many of the advantages of fully customized LSI without such drawbacks as high cost and long development time.

Today, gate arrays are rapidly becoming competitive in the same applications as PLDs. However, PLDs provide several advantages over gate arrays that make them much more amenable to customer needs for many applications.

The primary advantages of PLDs are their immediate availability, the lower risk they entail, and their lower overall cost per application. A design engineer, upon completing his design, can immediately customize a PLD to his specifications. If he discovers that he has made a mistake in his design, he can correct it and program another device. Since he customizes a part just before it is used, he undertakes no risk in committing to a particular design. If he needs to make a change, he can program a part from inventory to the new specification—no parts are wasted. He can program the new part in minutes.

According to a survey that appeared in the January 12, 1984, issue of *EDN*, the two greatest disadvantages of gate arrays are their high development cost and the lack of second sources for them. PLDs, in contrast, have an extremely low development cost and offer multiple sources. The short, simple development cycle for PLDs translates into easier use, lower costs, shorter time to market, and greater flexibility. Table 1 rates standard, semicustom, and full-custom integrated circuits according to various criteria used in choosing an IC for a design. A "1" indicates the best choice; a "4" indicates the worst.

Dataquest, a market research firm, lists in the December 28, 1984, issue of its *Nielsen Dataquest Research Newsletter* ("Gate Array Impact on the ASIC Marketplace," by Katy Guill and Andy Prophet) five needs of the gate array user, all of which are met efficiently by PLDs. The first is for quick delivery of prototype units. With PLDs, prototypes are available in minutes once the design is complete. The second need is for easy-to-use design automation tools that require a minimum amount of time to learn. PLD software tools include assemblers and higher-level languages that enable a design engineer to automatically convert whatever description he has created into a fuse plot. The PLD can then be programmed as easily as a PROM.

The third need is for prototype units that work the first time. PLDs, being standard parts, are fully tested during production and can be quickly verified after customization. If a design error has been made, the design can be quickly modified and another prototype can be programmed in minutes. Using personal-computer-based software and a standard programmer, the design engineer can stay at his own desk and proceed to alter his design on the PC, and then instantly download the new design to the programmer and the new device. The device can then be dropped into the target application and the design verified. The ease with which PLDs can be verified encourages experimentation with designs.

A fourth need is for low development and production costs. PLDs have extremely low development costs—only the up-front costs of the software and programmer. Their design cost consists simply of the design engineer's time. The production costs for PLDs are driven down by the multiple sourcing and high-volume production characteristic of

standard products. Gate arrays provide neither of these advantages.

The fifth need is for convenient design centers and training. The PLD design center is very convenient indeed—the engineer's personal computer. Our company, Monolithic Memories, offers tutorial aids in several formats. Moreover, the standard nature of the PLD makes it easier to understand and use than the more customized and thus less "friendly" gate array.

There are many other factors that influence the choice of semicustom device technology, of course, such as architectural efficiency. The ease of use of the PLD unfortunately leaves it much less efficient architecturally than other forms of semicustom logic. In a gate array, the user constructs logic functions using two-input NAND gates. In a PLD, the user constructs logic functions using higher-level macrocells. Naturally, there will be more wasted silicon in the PLD, although its optimized macrocell and efficient fuse array will still provide an effective structure. It is the gate array's very architectural efficiency that is the source of its high development cost, since that efficiency derives from computerized trial-and-error routing and a custom mask, both of which are expensive.

Figure 1 gives the 1985 sales volume for various types of ICs and estimates the 1990 volume for them. All types show healthy growth. Because standard cells are starting at a low level and because they are expected to become very popular as the IC market matures, they are expected to show the highest growth rate. Linear arrays, though relatively new and technologically difficult, should also show impressive growth. PLDs and gate arrays should also show very healthy growth, with the growth in PLDs (367 percent) exceeding that in gate arrays (234 percent). Part of the greater growth of PLDs is attributable to their ease of use, but part is also attributable to the ease with which they can be reprogrammed—for circuit designs that are likely to change, programmable logic is the only practical solution.

It is very difficult to directly compare PLD cost and gate array cost—one is comparing apples and oranges. PLDs have a minimal development cost and a standard unit cost. Gate arrays, on the other hand, have a large development cost and a small unit cost. Moreover, it is very difficult to determine the PLD architecture that is equivalent to a particular gate array and vice versa. Currently available PLDs such as Monolithic's MegaPAL devices have as many as 5000 four-input NAND-gate equivalents. However, a 5000-gate gate array probably cannot replace a 5000-gate-equivalent PLD exactly, and such a PLD cannot replace a 5000-gate gate array exactly. Such replacement is very application-dependent. A MegaPAL array, for example, is limited to 64 inputs and 32 outputs. If a gate array has more inputs and outputs, the MegaPAL device cannot replace it. And if the gate array cannot duplicate a feature of the MegaPAL array—such as its large fan-in on its AND gates, for example—then the gate array cannot replace MegaPAL array. Even at lower gate densities, comparisons remain application-dependent.

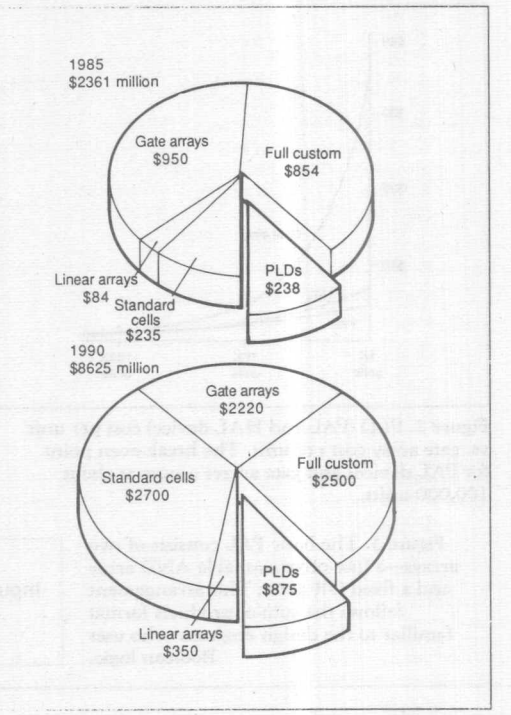


Figure 1. Growth of the worldwide application-specific IC market from 1985 to 1990. Dollars are in millions. (Source: Integrated Circuits Engineering Corp.)

For purposes of providing a *general* idea of the cost efficiencies over volume of gate arrays and PLDs, we can compare them as if they did have one-to-one equivalency, however. The nonrecurring engineering, or NRE, charge for a gate array can vary from $15,000 to $75,000, depending on complexity and design methodology. If a gate array has a $35,000 NRE charge and a $1.00 unit cost, its cost per unit drops almost inversely with volume, as shown in Figure 2. Now let us examine a specific type of PLD, the PAL device. If a PAL device costs $5.00 for a single unit and drops in price with volume according to standard pricing curves, it shows a more steady decline in price with volume (the PAL plot in Figure 2). The crossover point in this comparison is almost at 100,000 units. The graph shows quantatively the reasons why gate arrays become cost-effective over larger volumes, but it leaves out several other factors, including gate equivalencies, risk, and time to production.

Let us compare the design and production costs for PLDs and gate arrays. For a PLD, design cost means the design engineer's time, as we mentioned above. This is already a committed cost. However, for a gate array, design cost can

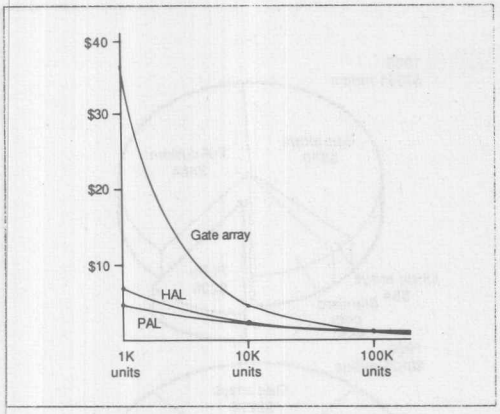


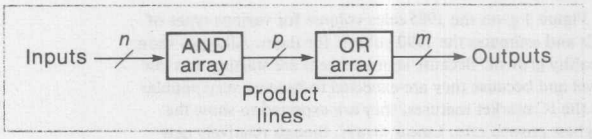
Figure 2. PLD (PAL and HAL device) cost per unit vs. gate array cost per unit. The break-even point for PAL devices and gate arrays occurs at about 100,000 units.

be a sum of several factors. The time a designer spends on a computer to create a design can be very expensive, and the software he needs to create and test the design can also be expensive. Production costs for both types of device include not just the cost of the silicon but also the cost of masks, fabrication, and labor. And if a device does not work as expected, the entire process must be repeated.

With a PLD, the only purchase cost is that for the silicon itself. The PLD is mass-produced, so all the supporting design efforts—fabrication, test generation, and reliability enhancements—are spread over time and many devices. The result is much lower cost to the user.

For high-volume, fixed applications, a PAL device can be replaced with a mask-programmed hardwired array logic, or HAL, device. The HAL device replaces the fuses of the PAL circuit with a final mask step that permanently metallizes the logic connections. This alternative is an excellent match for the volume applications that are attractive to gate arrays. The HAL circuit combines the lower unit cost of the mask-programmed logic device with the prototyping capabilities of the PAL circuit. It offers advantages unmatched by any other form of semicustom logic. Assum-

Figure 3. The basic PAL consists of two arrays—a fuse-programmable AND array and a fixed OR array. This arrangement follows the sum-of-products format familiar to the design engineer who uses Boolean logic.



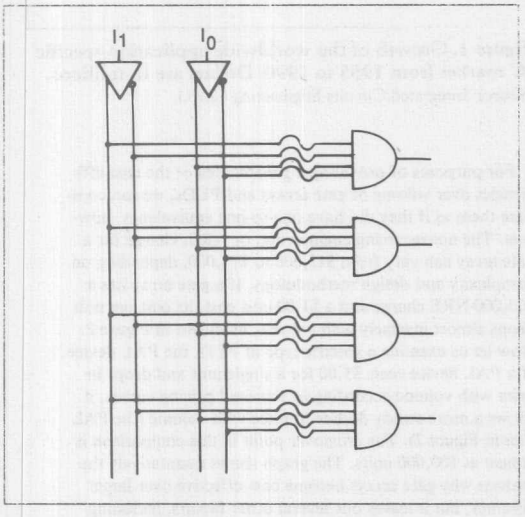


Figure 4. Two-input AND array with four AND gates. An input (in or /in) is disconnected from an AND gate when the fuse associated with that input is blown.

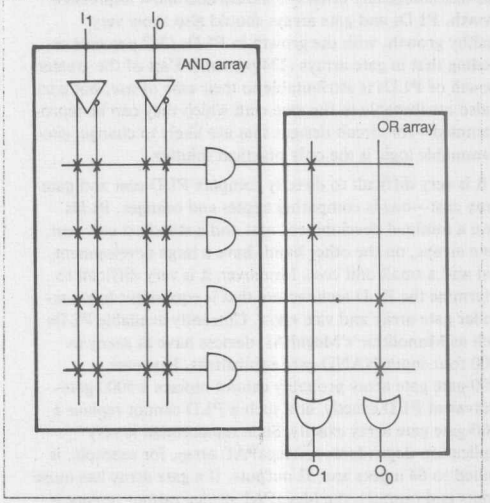


Figure 5. An "×" indicates an intact fuse. The two-input AND array with four AND gates (product lines) from Figure 4 is shown along with a two-output OR array. Note that the AND array is user-programmable, whereas the OR array is hardwired.

ing an NRE charge for a HAL device of $5000 and a unit cost of $2.00, we can plot the relationship of the HAL device to the PAL and gate array (see Figure 2 again).

PAL devices—historical background

PAL devices were introduced by Monolithic Memories in 1978. They consist of two arrays of logic gates. The first array consists of AND gates that form Boolean products of the inputs, and the second array consists of OR gates that sum these products (see Figure 3). Hypothetically, there are *n* inputs to the AND array and *m* outputs from the OR array. The AND array consists of *p* AND gates (and *p* product terms), and all *n* device inputs, both inverted and non-inverted, are potential inputs to any of the *p* AND gates. In an unprogrammed PAL device, every device input is actually connected to every AND gate. A device input ceases to be an input to a particular AND gate if the fuse that connects that input to the gate is blown. Fuses are blown according to a fuse map that is generated by the user via software. Figure 4 shows a simple two-input array with four AND gates. Figure 5 shows a simplified notation for this circuit—an "×" is used to indicate that a fuse is intact, and a single line is used to represent all lines of connection with an AND gate. (Note that the use of the single line does not imply the connection of those lines to each other.) Figure 5 also shows a two-output OR array that uses the same conventions for intact fuses and lines of connection. Most PAL devices have a considerably greater number of inputs, product terms, and outputs than shown here, but this example illustrates the general PAL structure.

The arrangement of the AND and OR gates follows the sum-of-products format familiar to the design engineer who uses Boolean logic to express his logic requirements. As implied above, the engineer can program the AND array in a PAL device by using links similar to those used in a PROM circuit. He opens the particular links needed to configure the logic to the user's specification.

Before discussing the PAL20RA10 in particular, we should note the differences between PAL devices and other PLDs. Most PLDs consist of an AND array followed by an OR array. This is also the architecture of memory devices, including bipolar PROMs. In a PROM, the AND array is fixed and completely decodes the inputs. The OR array is programmable and holds the information at each decoded location, or the word at every address (Figure 6a).

The first PLD was developed by Signetics and is called a field-programmable logic array, or FPLA. The FPLA is an implementation of the general PLD logic combination—an AND array followed by an OR array—in which fuses are used for interconnections in each array. The FPLA allows both arrays to be programmed (Figure 6b) and thus pro-

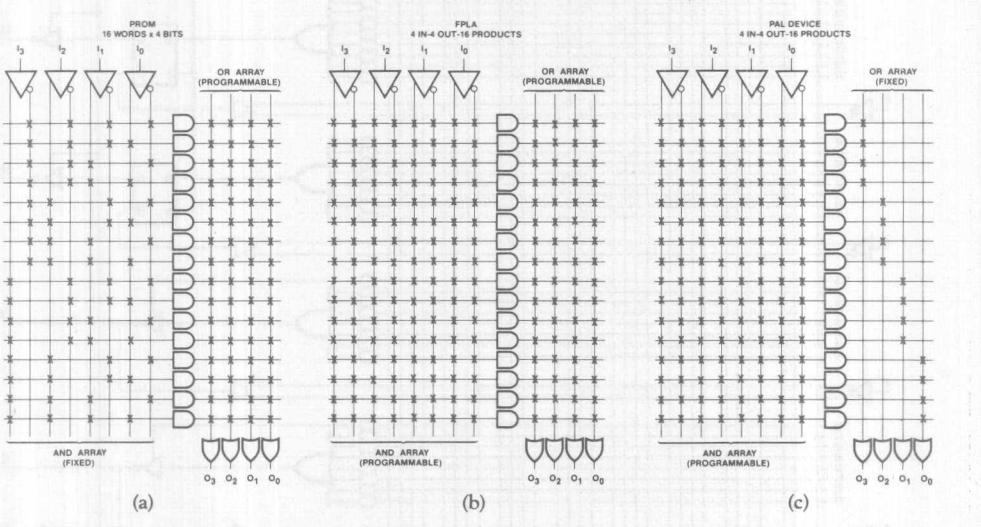


Figure 6. PROM AND-OR array (a)—the AND array is fixed to decode the address, whereas the OR array is programmable for the stored data; field-programmable logic array, or FPLA (b)—both the AND array and the OR array are programmable; PAL device (c)—the AND array is programmable whereas the OR array is fixed. The PAL device offers much of the flexibility of the FPLA along with the low cost and easy programmability of the PROM.

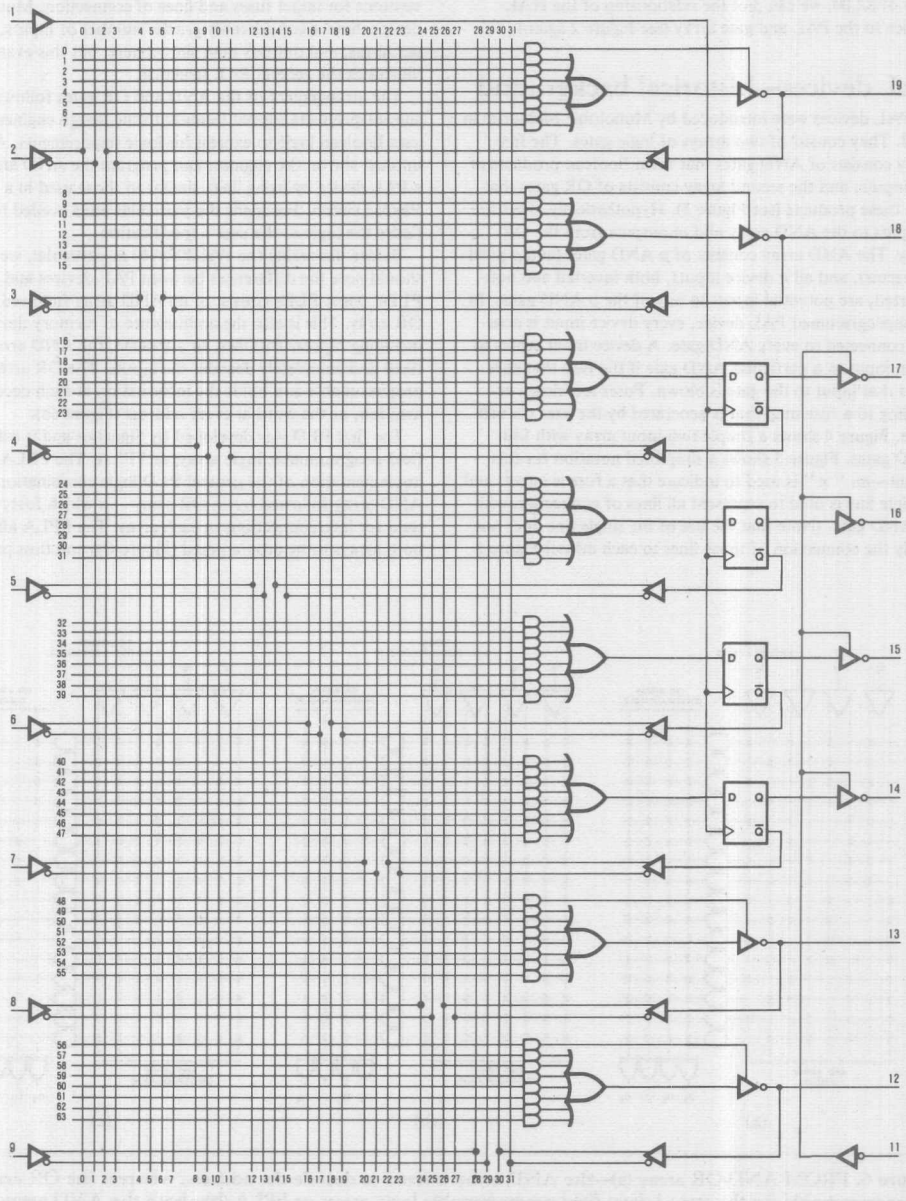


Figure 9. PAL16R4 logic diagram. This device has four registered outputs with a single (synchronous) clock.

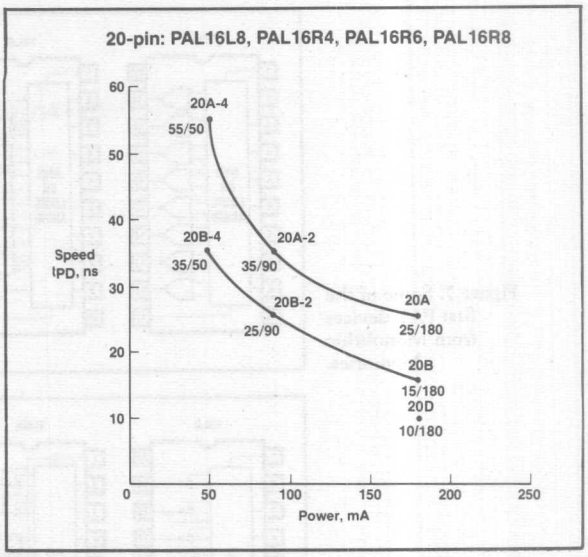


Figure 10. History of the speed improvements and power consumption reductions for devices in the Medium PAL family.

New product development

New product ideas in semiconductor companies come from a variety of sources—from customers and those in direct contact with them to design engineers who have ingenious ideas about how to add features without adding cost, power, or timing delays. Since customers are usually willing to describe the product that would be ideal for them, they are often a good source of ideas. Product ideas usually pass through the following steps.

First, the idea is generated. Typically this is in response to a limitation in a current product and is an evolutionary step. Revolutionary ideas such as the original PAL concept are rare and usually come from approaching old problems in a new way. If the idea is a customer's suggestion, it is usually reported to the factory in a trip report.

The idea, whether generated internally or by a customer, is next submitted to a group typically consisting of representatives from product planning (including someone from the product area into which the idea falls, whose job is to examine such ideas), marketing, design, and product engineering. The group discusses the new idea and responds to it; it either drops the idea, giving reasons for doing so, approves it for further consideration, or asks for more details.

If a product idea is approved for further consideration, a member of the product planning group carries out market surveys and analyses. He contacts customers, often in person, and locates third-party market studies and research. He analyzes internal capabilities and seeks out information about products that could compete with the proposed product. At most companies, the result of his efforts is a new product approval document consisting of a description of the product and its applications, alternatives to the product, a discussion of the market for the product, and a marketing strategy for it.

If the product is approved by all the representatives in the group, it is assigned to design engineering. The design engineer is free to use whatever tools he chooses within the definition set out for him. This definition includes such items as performance requirements but rarely specifies the process technology. It is often modified as new requirements arise.

Design engineering is finished with the product only after it has been successfully produced on the production line. It is then released to product engineering, which takes on full responsibility for the product for its lifetime. Product engineering is typically involved before this point, however, making sure the product is progressing well and problems are being eliminated quickly.

The final step is release to the marketplace. For most companies, release to market means sending out data sheets and samples to the field once production quantities have become available. For programmable products such as PAL devices, software and programmer support must also be in place.

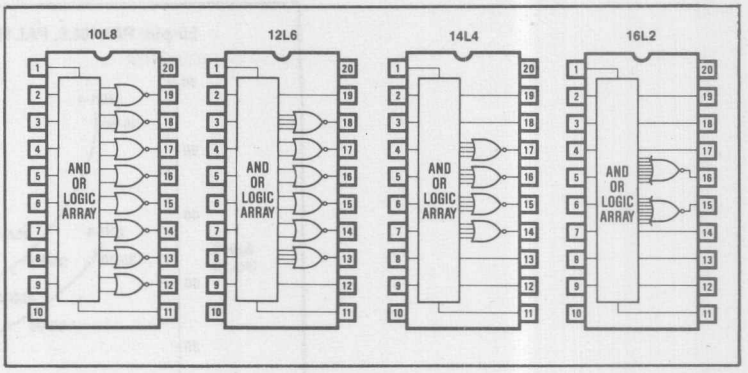


Figure 7. Some of the first PAL devices from Monolithic Memories.

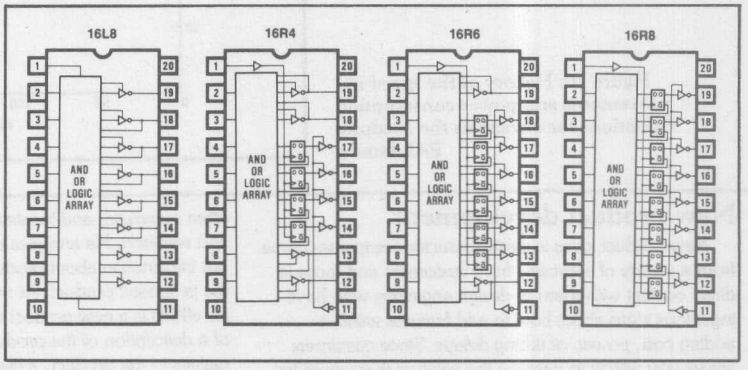


Figure 8. The first registered PAL devices.

vides a great deal of flexibility. However, its programming circuitry and fuses lead to a large, slow, power-consuming device and make programming expensive and difficult.

In 1977, John Birkner of Monolithic Memories got the idea of taking the FPLA architecture and hardwiring the OR array, leaving only the AND array programmable (Figure 6c). The result was called a PAL circuit, PAL being the acronym for programmable array logic. (Monolithic Memories has since registered PAL as a trademark describing its products.) A PAL device offers higher speed and consumes less power than an FPLA. The first PAL devices were designed so that engineers could program them just like 512 × 4 PROMs, using phantom fuses to fill in beyond the 512 fuses in the actual devices. Thus, engineers could program them on widely available, inexpensive programmers.

The first family of PAL devices included both combinatorial and registered devices. Combinatorial devices consist of just a programmable AND array followed by an OR array, with no registers or other macrocells. They are fairly simple and provide little more than a varying number of inputs and outputs (see Figure 7). Registered devices, on the other hand, provide clocked registers for the outputs and allow the previous output state to be stored and used for determining the next state. These types of devices are much more challenging. A register allows the device to be used as a state machine, one in which the outputs depend not only on the present inputs but also on the previous state.

Since a state machine can be implemented in a variety of forms, the registered PAL architecture is difficult to define. To do so, one must specify the total number of registers, whether a register can feed its output back into the logic array, the nature of the control of the output enable following the register, and common register features such as preset and clear functions. These specifications have a major influence on the potential applications of the device.

Monolithic's original registered PAL product family, whose members were designed to be used as state machines, is the Medium 20 family, where Medium describes the logic density and 20 is the number of pins. The family consists of four devices, all with 16 complemented inputs to the AND array and eight outputs from the OR array. They have varying numbers of registers—zero, four, six, or eight—which is reflected in the part numbers, PAL16L8, PAL16R4, PAL16R6, and PAL16R8 (see Figure 8). The 16R4 directly implements Mealy state machines, whereas the 16R8 directly implements Moore state machines. In the 16L8, 16R4, and 16R6, combinatorial outputs are enabled by individual product terms from the AND array. The product term enables allow the combinatorial outputs to function as inputs. These inputs, along with feedback from the registers and eight dedicated inputs, provide the 16 inputs to the AND array. The logic diagram of the PAL16R4 is shown in Figure 9.

The architecture of this family has proven capable of

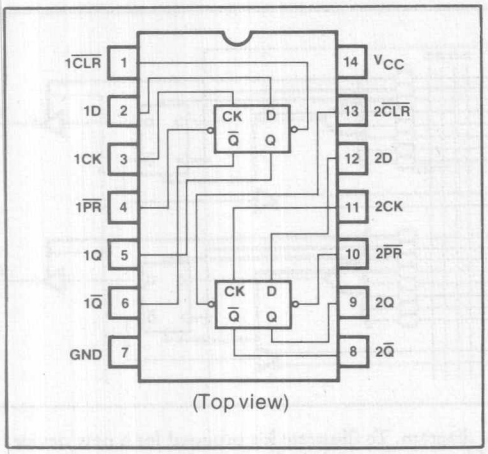


Figure 11. 7474 dual D-type flip-flop. Note the two separate clocks that allow asynchronous operation of the registers.

handling a remarkable percentage of what customers require in programmable logic. The improvements customers typically request are higher speed and lower power consumption. We have made progressive improvements in speed, to a 10-ns propagation delay in the Medium 20D Series and a 6-ns one in the ECL PAL IC. We have also reduced power consumption without sacrificing speed. The 20B-4 Series requires one fourth the power of the original 20 Series yet provides the same speed. With a propagation delay of 35 ns, the 20B-4 devices have a maximum supply current of 55 mA, compared to 180 mA for the original parts at the same speed. CMOS mask-programmed versions requiring zero standby power are also available. Figure 10 shows the history of speed and power consumption improvements in the Medium 20 family.

While we have been making these improvements, we have also introduced new families with higher logic densities and the 40- and 84-pin MegaPAL families. We have added new features to these families, including additional logic capabilities such as exclusive-OR gates, arithmetic feedback, programmable output polarity, product term sharing, register bypassing, and register preloading. We made these enhancements in response to specific customer needs, and they have opened new applications for programmable logic.

Most of these new features were simply extensions of the original PAL concept. The original PAL family was a breakthrough, a new alternative offering completely new advantages in semicustom logic. In building on this family, we concentrated on speed and power improvements and not on finding revolutionary new architectures. However, the PAL20RA10, which we will discuss shortly, is another breakthrough concept. It did not derive from extensions to the original idea but arose as a new response to the needs of customers using PAL devices.

The PAL20RA10

The PAL20RA10 was first proposed at an internal Monolithic Memories conference in 1981. The original concept was described as an "unstructured PAL" having independent programmable clock, reset, and set controls on each flip-flop. Until then, all PAL devices had been structured—or synchronous—in architecture; all transitions on register outputs occurred simultaneously in response to the rising edge of the clock signal. The proposed ability to control the clock independently for each flip-flop would enable the engineer to create asynchronous logic within a PAL chip.

The idea was proposed by Harry Hughes, who was then a field applications engineer, or FAE, in Great Britain. FAEs are the company representatives who have primary contact with customers; they have responsibilities ranging from helping existing customers with complex design problems to introducing Monolithic Memories to potential customers. As the primary customer contact, they are quite familiar with real user needs.

The concept of the unstructured PAL circuit had originated with several European customers. They had been using the Medium 20 PAL Series in large quantities. However, the application logic they had been integrating had not always fitted well in PAL devices. They had often had sections of individual 7474 flip-flops mixed among random logic. A 7474 device (see Figure 11) consists of two independent D-type flip-flops, each with its own data input and complementary outputs, and clock, preset, and clear. A standard PAL16R8 has eight of these D-type flip-flops but only one clock and no preset and clear functions. Thus, integrating the collection of 7474s and the glue logic had required the customer to convert a multiclock design into a single-clock one. This had been difficult or impossible to do with standard products.

The customer requested an independent clock, set (preset), and reset (clear) for each flip-flop. Since the FAE had found other customers requesting similar architectures, he proposed the device. He took the 16R8 logic diagram and added the four control terms, as shown in Figure 12. He described the concept at Monolithic's FAE conference in July of 1981.

FAE conferences are held to foster technical exchange between the factory and the field. Problems are identified and ideas are discussed. Now held every six months, these conferences are critical for planning future product directions.

After it had been presented at the FAE conference, the unstructured PAL circuit was discussed within the factory. For an idea to go somewhere, factory discussion must produce a business plan for the product that describes it completely and estimates the profits that can be expected from it. New product ideas must pass several stages of analysis and approval. (See "New product development," at left.)

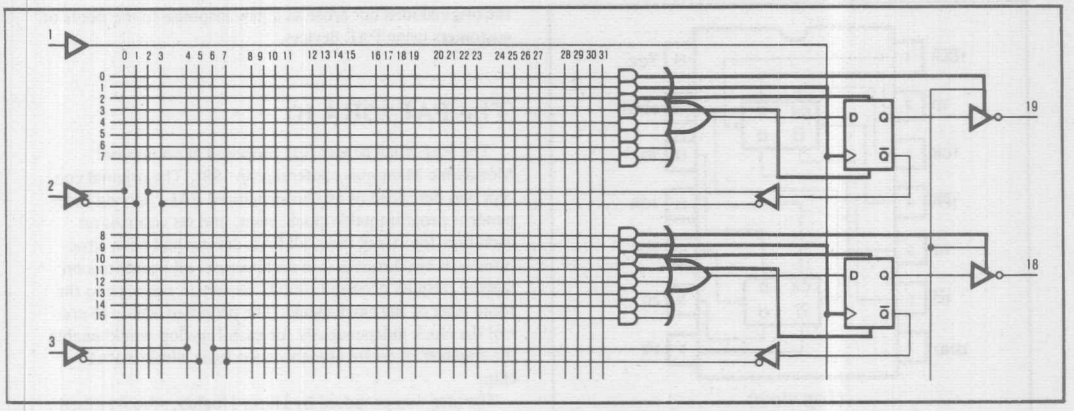


Figure 12. Harry Hughes' modification of the PAL16R8 logic diagram. To illustrate his proposal for a new device, he added four product terms (heavy lines) to control the enable, set, reset, and clock.

The major problem with the unstructured PAL circuit was its inherent lack of testability. The power its architecture gives the customer also demands that the customer use it carefully. For example, the minimum setup time required before the register can be clocked must be carefully considered when the clock signal is a complex product term. Data must be made available to a flip-flop early enough so that even the minimum time to activation of the clock satisfies the setup time required. This required delay can be very difficult to calculate, especially if the clock term is under the control of another asynchronous register. All of the internal asynchronous signals must be handled with care to prevent invalid combinations.

These and other testability issues caused great concern and led to several unique solutions. Internal test features were added, including the ability to test both DC and AC characteristics on unprogrammed devices and even the ability to test programming characteristics without affecting the fuses in the actual array. Two external pins were given dedicated functions: one is an output enable that allows the device to be disabled during board-level testing; the other is for register preloading, which allows the user to gain access to the flip-flops without going through the fuse array.

The final, approved product was called the 20FF10 at first. It had 20 array inputs and 10 outputs, and enough new features to earn it the name "Feature Freak"—the source of the "FF" in the designation. However, a name must uniquely identify a product and must do so in terms the customer can recognize. The "FF" for "feature freak" was unique but not very meaningful to the customer, so this part of the name was changed to RA, for "registered asynchronous," and the new device became the 20RA10.

The approval document for the device described its architecture, which is shown in Figure 13. The clock, set, and reset are controlled by product terms, as shown in Hughes' original sketch of the unstructured PAL circuit (see Figure 12 again).

A compromise was made for the output enable. An external output enable is required for most applications, especially for disabling the device during board-level testing. But an internal (asynchronous) output enable for independent enabling of the flip-flops is needed, too. One solution would have been an external enable pin for each output, but that would have been a highly inefficient use of external pins. The solution that was used is an enable signal that is the product of the external common enable pin and an internal independent product term. Thus, the enable can be activated either externally (in common) or internally (asyn-

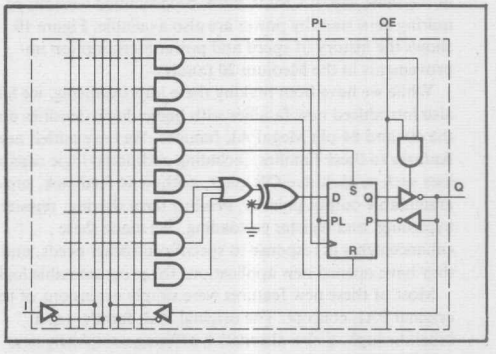


Figure 13. The PAL20RA10 output cell. Note the separate product terms that control the set, reset, enable, and clock. Note also the output enable that can be controlled either externally (synchronously) or internally (asynchronously).

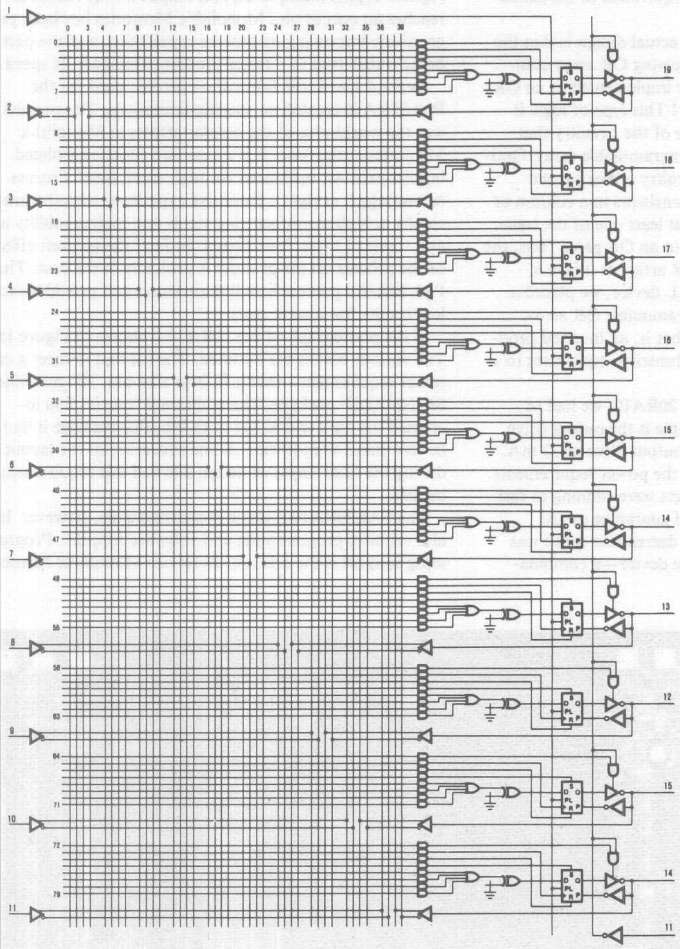


Figure 14. Logic diagram for the complete PAL20RA10 circuit—AND array, OR array, and asynchronous registered outputs.

chronously), satisfying both the design engineer, who wants the internal enable, and the test engineer, who wants the external control.

Another relatively new feature was implemented in this circuit—programmable polarity. This allows the output polarity to be either active low or active high. The design engineer can then write logic equations either way, allowing his software to match the sense of an equation to the actual device. This enables the designer to fit more logic into a single PAL circuit.

The problem of how many registered and combinatorial outputs to provide had been solved in previous devices by offering a family of combinations. The solution to this problem for the PAL20RA10 came somewhat by accident. The situation of set and reset going active simultaneously had to be dealt with. We decided that allowing this situation to result in bypassing of the register provided a beneficial new feature. If both the set and reset product terms are activated, the register becomes transparent and the output becomes combinatorial. Because of this feature, the logic designer can decide at programming time how many registers he wants (from zero to ten) and on which outputs he would like them. This architectural flexibility is unique within programmable logic.

These features are shown in the logic diagram in Figure 14. However, though they describe exactly what the user en-

counters, they do not actually exist within the device itself. The logic diagram is only a logical equivalent to the actual layout of the circuit.

The first thing to note about the actual design is that the programmable AND array and following OR array common to all PAL circuits are actually implemented as an OR array and a following NAND array! This type of logic is required because of the architecture of the circuitry that senses the ones and zeroes in the programmable array. Optimized for bipolar PROMs, this circuitry senses current flowing from one or more of the transistors in a column of the array. Since it senses current if at least one of the transistors is on, the logic is equivalent to an OR gate. Thus, the programmable array must be an OR array, as it is in a PROM. To create the logic of a PAL device, we placed a logical NAND array after the programmable OR array, creating an OR-NAND structure, that is, an inverted product of sums, which by Boolean arithmetic is equivalent to a sum of products.

To maintain the flexibility of the 20RA10, we had to make various trade-offs. One example is the output drive. The original proposal specified an output drive of 32 mA. We lowered this to 8 mA to reduce the power requirements.

Although standard 24-pin products were running as fast as 25 ns, the highest speed we could guarantee on the PAL20RA10 was 30 ns. This slight decrease in speed was due to the additional features of the device—a combinatorial output signal travels through the polarity control, register bypass multiplexer, and output buffer before it reaches an output pin. Monolithic Memories has been progressively improving its process capabilities, and the part is being transferred to a faster process to increase its speed.

Beyond speed, the fabrication process used for the PAL20RA10 created no specific limitations. The process was the workhorse of the bipolar efforts at Monolithic Memories at the time. It is a standard junction-isolated bipolar process, optimized for high speed (see "Process technology," at right). Extensive experience with the process results in high manufacturing yields and high reliability and, as a result of these, lower costs. In fact, the greatest effect of the process on the product is probably in the cost. The PAL20RA10 process is well understood and provides excellent results for what it costs.

A die photograph of the 20RA10 is shown in Figure 15. The various blocks are outlined. The die had to meet a customer requirement—that it fit into a 24-pin, 300-mil-wide SKINNYDIP package. The additional features had increased the die's width, yet to fit into this package it had to be less than 140 mils wide. Some squeezing of the layout during the final stages of design allowed this requirement to be met.

The PAL20RA10 is much more than a die, however. It also requires programming and software support. Programming support is the one critical service Monolithic Memories

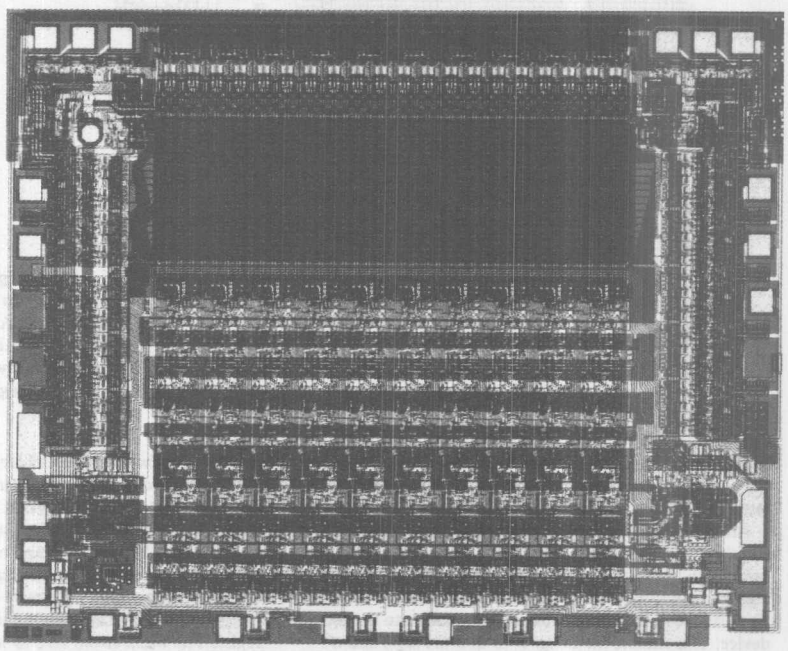


Figure 15. PAL20RA10 die photo.

does not provide, and thus it depends on the major PLD programer manufacturers for it. But to be able to support a product at market release, the programmer manufacturers must be provided with the programming specification for the product months before it is released. Monolithic therefore keeps in close contact with these manufacturers.

Software support is provided by Monolithic Memories through its PAL assembler software package, PALASM 2. PALASM 2 takes Boolean logic equations, tests them against expected outputs, and converts them to a fuse map for a PAL device. A description of this fuse map can then be downloaded to a programmer in a standard JEDEC format for customization of the chip.

Software support is a critical part of the product because it is through software that the customer implements his design. A designer wants the conversion from what he knows to be the required logic to what a PAL programmer can understand to be as simple and easy as possible. PALASM 2 allows the designer to express his function with Boolean equations. It then uses that input to configure output polarity, register bypassing, and all the other configuration features.

Process technology

The process used for the PAL20RA10 is a junction-isolated one. A cross section of the 20RA10 is shown in Figure A. NPN transistors are separated from one another by junctions.

The process is a washed-emitter one. In this process, a layer of nitride (Si_3N_4) is used as a mask. This mask layer defines the size of the emitter and the emitter contact and allows the emitter and the emitter contact to be the same size. In other processes, the emitter is usually made larger than the emitter contact to guarantee good contact. When formed at the same time, however, they can be the same size and still maintain proper contact. This allows the emitter to become the minimum feature size of the design, providing a smaller overall die size than the die size that could be provided by a standard process.

A programming fuse is located between an emitter and a product line that in turn leads to a collector (Figure B); driving a large current through this fuse opens its narrow neck and causes it to change from a very low resistance to a very high one. The process does not cause the fuse to explode, as some claim; it merely severs the greater part of the connection. This change opens the link between the emitter and the product line.

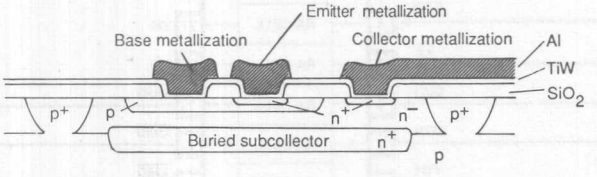


Figure A. Cross section of the PAL20RA10—NPN transistors are separated from one another by junctions.

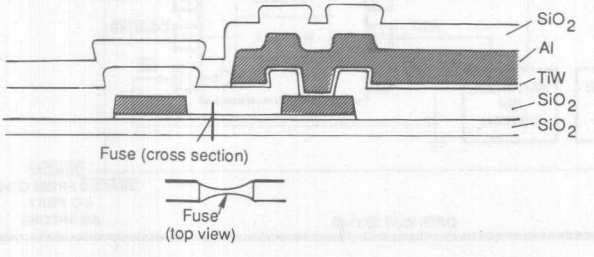


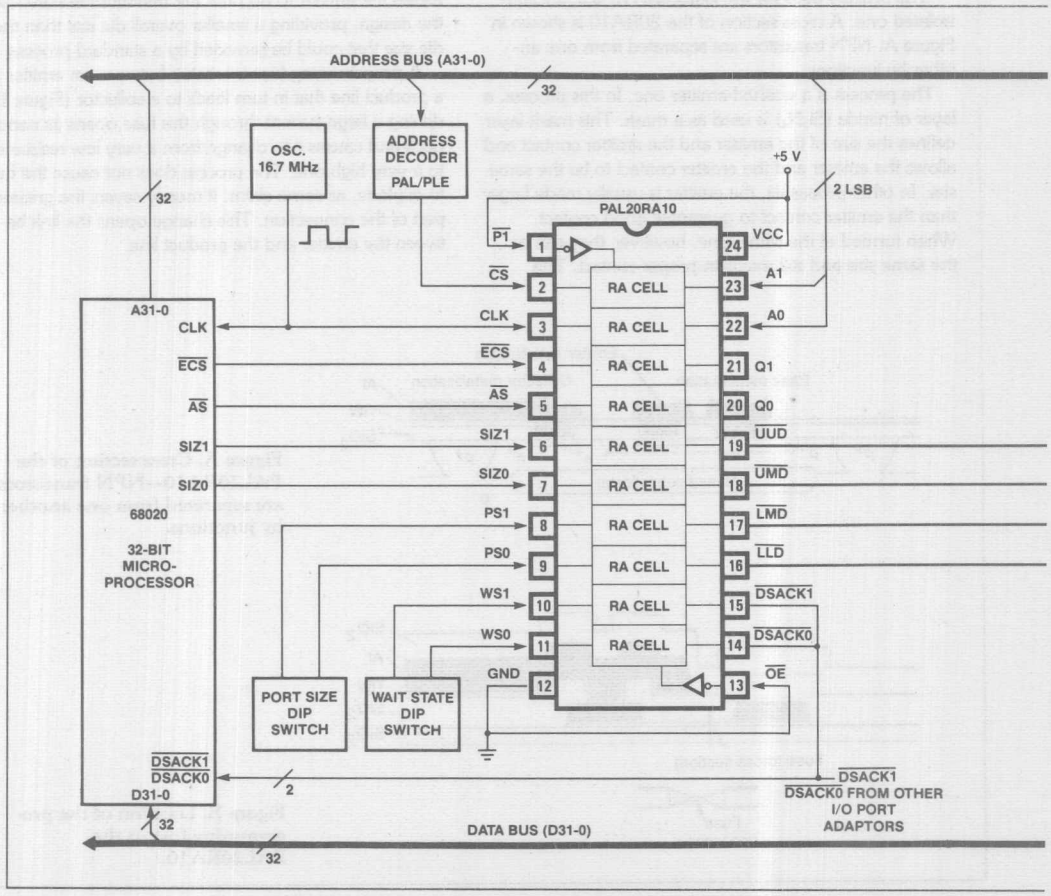
Figure B. Location of the programming fuse in the PAL20RA10.

The new features of the PAL20RA10 required that some new syntactical conventions be added to the PALASM 2 software package. In addition to the standard logic transfer functions, the designer has to be able to specify several novel architectural situations. For example, he has to be able to specify register bypassing in either of two ways: implicitly or explicitly. He specifies bypassing implicitly by writing a combinatorial logic equation using an "=" sign instead of the ":=" sign, which is reserved for registered outputs. He specifies bypassing explicitly by activating the set and the reset on the flip-flop simultaneously, causing the register to be bypassed and the output to be combinatorial.

The control terms are described by dot operators such as "term.set." We selected dot operators for the same reason they are used in popular word processing programs—they rarely occur in natural language and can therefore be given special significance.

Third-party software is available that allows other forms of input such as schematic entry. ABEL from Data I/O Corporation and CUPL from Assisted Technology both provide high-level support for PLDs.

As we asserted earlier, true customization of the PAL20RA10 involves much more than matching the function to the customer's requirements. Speed and power consumption must be acceptable, design support for the device must be available, pricing must be reasonable, product availability must be high, and quality must be high. The PAL20RA10 has been in full production for well over a year, and to the good fortune of Monolithic Memories all of these requirements have been met. Moreover, FAEs are ex-



cited about teaching customers about a product that originated with one of their own members. In fact, the 20RA10 was the first product proposed by an FAE to go into production at Monolithic Memories. Market acceptance has been beyond expectations, and as a result FAEs are being brought even further into the product planning process.

The PAL20RA10 is the first member of a new family of devices; new products are in various stages of proposal, design, and production. A 20-pin spin-off, the PAL16RA8, is now in full production. A higher-speed version is in design. A version in a 40-pin DIP was considered, but since 20RA10 outputs operate independently, the higher density and resulting larger package were difficult to justify. Two 24-pin, 300-mil-wide parts can do more in less board space than one 40-pin, 600-mil-wide device.

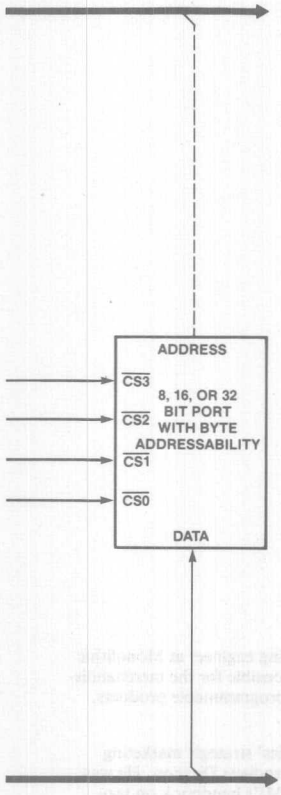


Figure 16. A PAL20RA10 used as a universal port interface adapter for a Motorola MC68020 32-bit microprocessor. It generates logic that maintains both byte and word addressability within the 68020's 32-bit-wide word format and that makes possible the use of both high- and moderate-speed components and peripherals in a single system.

The PAL20RA10—an application example

To demonstrate the capabilities of the PAL20RA10 and the PALASM-based design process, we present an application that uses the PAL20RA10 as a universal port interface adapter for a Motorola MC68020 microprocessor (Figure 16). This example should help demonstrate the features we described above and point out how easily the 20RA10 can be customized to a particular user's application.

The design engineer began this design by considering the large blocks: the microprocessor, the memory, and the I/O. Now, near the end of the design, he needs to place some glue logic between the microprocessor and the I/O ports. He is open to anything from standard to full-custom logic, and he considers each alternative.

Because he is very unsure of the finality of the design, he does not want to commit himself to a major development effort. Therefore, he rules out full-custom logic and standard cells. He may be able to make a gate array work, but he does not have time to work with a foundry in specifying the design nor time to wait for the foundry to deliver the part. In fact, he would like to have the part immediately. Moreover, since his design is not yet proven, he cannot justify the high cost of a gate array development effort.

He could use off-the-shelf ICs easily enough, but he does not want to take up that much space on his board. He has a limited amount of room with which to work. He could use standard LSI devices, but none provide the functions he requires. Thus, he arrives at the programmable logic device as the logical alternative.

As we said above, in this design the PAL20RA10 functions as an asynchronous universal port interface adapter for a 32-bit microprocessor, the MC68020. Among the greatest challenges the engineer faces when designing with the new generation of 32-bit, high-performance microprocessors is maintaining both byte and word addressability within the microprocessor's 32-bit-wide word format and accommodating both high-speed and moderate-speed memory components and peripherals in a single system.

The MC68020 has a dynamic bus sizing feature that helps solve the first of these two problems—it has internal logic that manipulates and directs words smaller than 32 bits. However, it needs external logic to handle 8- and 16-bit memory arrays and peripherals. It even needs such logic to maintain byte addressability of 32-bit memory arrays. This external logic must be able to decode the transfer size control signals from the MC68020 so it can generate memory chip selects and handshake back to the microprocessor with the port size acknowledge signal. The PAL20RA10 is well suited to this task.

External logic also helps the MC68020 solve the second of the two problems. The 16.7-MHz speed of the MC68020 version used here translates into 60-ns machine cycles and 90-ns memory cycles, and therefore the MC68020 can keep up with many of today's fastest memories. But it may have to work with slower bulk memories as well. To do this without slowing down the high-performance memories, the

Table 2.
DIP switch settings used in the universal port interface adapter to choose the data bus port size.

Position PS1	Position PS0	Data bus port size
L	L	No port present
L	H	8-bit
H	L	16-bit
H	H	32-bit

Table 3.
DIP switch settings used in the universal port interface adapter to choose the number of wait states.

Position WS1	Position WS0	Number of wait states
L	L	0
L	H	1
H	L	2
H	H	3

68020 must be able to insert wait states into the memory cycles of the slower memories. The 20RA10 can help the 68020 do this by intercepting microprocessor memory access requests and, when needed, generating control signals to the microprocessor requesting wait states.

In this design the PAL20RA10 functions as a universal port interface adapter. Both blocks of external logic we alluded to in the previous two paragraphs are implemented in a single PAL20RA10. The asynchronous capabilities of this PAL circuit are required by this design because the MC68020 provides asynchronous control signals that must be processed by the PAL device between clock signals. On the other hand, the synchronous capabilities of the 20RA10 are needed for generating memory control signals. An internal PAL state machine must be synchronous with the system clock. Therefore, neither a combinatorial (i.e., asynchronous only) nor registered (i.e., synchronous only) PAL device will fill the bill. The PAL20RA10's unique combination of features allows it to provide a single-chip solution.

The PAL20RA10 used here generates the byte data select signals (UUD, UMD, LMD, and LLD) and the current port size select signals (DSACK0 and DSACK1) and performs wait state generation. The designer selects the data bus port size and the number of wait states by setting DIP switches (Tables 2 and 3).

The PAL20RA10 is a breakthrough product. It does not merely extend existing capabilities; it adds novel and highly useful features. It is also the first product produced by our company that originated with a customer. Its technical and market performance has exceeded our expectations and has proven to us the wisdom of listening to customers. As a result, we have devised ways of listening to them even better.

Questions about this article can be directed to Marc A. Baker, Strategic Marketing, Programmable Products Division, Monolithic Memories, Inc., 2175 Mission College Blvd., Santa Clara, CA 95054.

Marc Baker is a strategic marketing engineer at Monolithic Memories, Inc., where he is responsible for the merchandising and document literature for programmable products. He holds a BSEE from Stanford.

Vince Coli is Monolithic Memories' strategic marketing manager in the Programmable Products Division. He was one of the original authors of MMI's handbook on programmable devices. He holds a MSEE from Santa Clara University.

PLDs Abound: RAM-Based Logic Joins In

CMOS Logic Cell Arrays, only recently available from more than one source, are almost as useful as small gate arrays, but don't carry the risk of commiting a design to silicon.

By Marc A. Baker and Chris Jay
Product Applications
Monolithic Memories Inc.
Santa Clara, Calif.

The rise of advanced CMOS technology has dramatically affected the PLD market. Although not as fast as bipolar devices, CMOS products are gradually approaching the performance that was expected of bipolar about two years ago.

The logic designer gains two significant benefits from the latest in CMOS: greater densities for more complex logic, and reduced power consumption at low clocking rates. Since cutting power consumption means smaller power supplies and heat sinks, both factors reduce the size of electronic equipment.

Most CMOS parts can be backed up by batteries, making them useful for RAM-based devices such as the Logic Cell Array (LCA).

Although the LCA is based on a dense RAM architecture, the array itself is not a conventional RAM-based device. Actually, it's a PLD, but the dense CMOS RAM enables the designer to configure logic cells within the device and to specify I/O blocks. The interconnected logic cells, called CLBs (configurable logic blocks) and IOBs (input/output blocks) allow complex circuits to be constructed on-chip. And the LCA's battery backup powers down to save data.

Its dense architecture enables complex VLSI functions to be programmed in the cell array. Also, the random glue logic that would otherwise ride on system cards is incorporated into the array. Because the LCA is RAM based, it must be configured on power-up from a non-volatile source. Configuration data can be read from a CMOS EPROM on a byte-wide basis. When the reading is done, the EPROM is switched out of the circuit by deselecting its chip select input. Large EPROM devices may store three or four configurations, and tying the higher-order addresses to DIP switches allows the user to select one of a given number of logic functions. Thus one card may carry a large EPROM and an LCA, allowing for some degree of deferred design.

The LCA is actually a PLD, but its dense CMOS RAM enables designers to configure logic cells within the device and to specify I/O blocks.

Dual-layer metal, advanced twin-well CMOS has produced a programmable logic device that can extend the PLD into the area of low-end gate arrays.

Any custom or semicustom device must be tested, and PLDs are no exception. But since they are only functionally configured, functional testing offers a high degree of reliability. Such testing can be done with the programmer, immediately after programming. Production volumes can be taken care of simply by using an automatic handler.

An alternative to user programming is factory programming: Manufacturers can furnish factory programming or can create custom masks for hard-wired versions of the programmed pattern.

PLDs also are less expensive than other alternatives. Examining chip price alone, both mass-produced SSI devices and custom parts look very good. But other costs, such as inventory handling and non-recurring engineering (NRE) charges must be taken into account.

One way to estimate the cost of such items as inventory, testing and board space is to add about $1 per chip. Thus, if five SSI chips cost only 25 cents each, they actually cost the user $1.25 each, or $6.25. Those five devices can therefore be economically replaced by a $5 PLD. Today, many PLDs cost less than the devices they'll replace, and prices will continue to drop.

A gate array can replace multiple PLDs, but creating a gate array demands a large NRE charge for, say, producing masks and generating test programs. The NRE charges for a large gate array, including design costs, masks, a possible design fix and test programs, can be tens of thousands of dollars. A PLD has no NRE charges.

With the LCA, testability becomes an interesting issue—since the device is RAM based, test programs can be loaded into it. Once configured for testing, test vectors may be applied

IMPLEMENTATION COST COMPARISON

Cost	PLD	SSI	Gate Array
Price	Medium	Very low	Low
Inventory & Handling	Medium	Medium	Medium
NRE	None	None	Very High
Design Time/ Equipment	Low	Low	Medium-High
Design Changes	Medium	Medium	High
Test	Very Low	Very Low	High
Quantity Required Per Application	Low-Medium	High	Low

SSI chips still cost the least per unit. But since more are needed, circuit size can be limiting. Gate arrays cost less than PLDs, but their high NRE can make the difference in the long run.

Copyright© 1987 by **CMP Publications, Inc.**, 600 Community Drive, Manhasset, NY 11030.
Reprinted with permission from Electronic Engineering Times.

to determine if the logic cells and I/O blocks are functioning properly. For example, when configuring an LCA with buried diagnostic shift registers, serial diagnostic data may be shifted through the system to a serial output port, where the data would be available for analysis. When a complete system is configured for diagnostic testing, internal nodes that were previously inaccessible become available via a serial diagnostic scan loop.

When the LCA serves as part of a larger system, it can be initialized with buried shift registers performing functional tests on the system itself. Once testing is complete, the LCA can be set up for the required function.

Suppliers

In 1978, only two suppliers were producing PLDs. Now there are more than a dozen, with several sources for the most popular parts. Higher-volume production and second sourcing alleviate concerns with factory-programmed devices, like gate arrays. Control of the design is returned to the customer.

In addition, the customer has choices not only in silicon but in service and support. Manufacturers have varying types of quality control, test procedures, reliability verification, software and technical support, price ranges and delivery capabilities. As a recent survey in *EE Times* showed, these issues are very important. Quality control was the primary thing wanted in a supplier, and software support was the second most important issue.

Telecom Applications

Because CMOS devices are not power hungry, they are welcomed in telecommunications.

Low power consumption in modems, fax machines and the like means more features and more functions without an increase in size.

Further, in central switching stations, units must be kept compact because many line interface cards are likely to placed in one rack, and many racks might share one cabinet.

The LCA can put its talents to good use in telecommunications, including time-division multiplexing, line encoding/decoding and corner bending circuits. Also, zero-standby-power PLDs are in the works for line encoding/decoding, terminal adapter interfaces, CRC error correction, frame synchronization and other applications.

Since the LCA is reconfigurable, it's possible to modify the logic after design. In telecommunications, that means that functions can be changed to accommodate different communications standards and protocols, or both, without resorting to separate logic circuits for each.

Tomorrow's Market

Tomorrow's market will be driven more by users rather than suppliers. Applications beyond computers will be found, and engineers will push for the required technology. Moreover, users will drive product definitions. Software will be an integral part of the design solution and will require more adaptation to a user's design environment. An important factor in the increasing role of the user in the PLD market is the fact that PLDs are becoming part of up-front system design, not just last-minute replacements of discrete logic.

New settings for PLDs in high-speed processors, superminis and military equipment are fueling advances in bipolar speeds. In the past, the fastest digital systems could not employ programmable logic, but today's PLDs are matching and surpassing their speediest rivals.

Since the device is RAM based, test programs can be loaded into it, and vectors can be applied to see if the logic cells and I/O blocks are functioning properly.

CMOS technology is being forced to meet the demands of telecommunications, as well as adapt to industry and instrumentation. Such equipment calls for low power and immunity to temperature, power and noise fluctuations. CMOS promises to meet those needs, although most CMOS PLDs today simply match low-power bipolar specs. Tomorrow's CMOS PLDs will offer the zero standby power, wider operating conditions and dc characteristics of standard CMOS products.

Specific Parts

Users are also demanding more application-specific PLDs. At first appearing to be redundant, application-specific PLDs are devices whose architecture is aimed at a particular application. Such architectures may boast bus interface controllers, signal generators and decoders. True, these devices are limited to specific tasks, but will serve better than their generic kin. Suppliers are taking risks in pursuing limited markets, but the clear winner is the user who will soon have devices tailored to his needs.

Recently, there's been an explosion in the number of PLD architectures and the amount of support software. The first PLDs were programmed by manually encoding the individual fuses, which then had to be blown. Later, software was supplied that automatically converted Boolean equations into programming patterns. Now, software can use a number of design descriptions (equations, schematic capture, truth tables, etc.) and can simulate the design and create test vectors in addition to the programming pattern.

A number of support packages are on hand to give the engineer the CAD facilities that ensure a successful design. Most of these packages play on an IBM-PC and start with assemblers that convert Boolean equations into fuse plots suitable for programming into the PLD.

More complex devices require greater levels of software support, so state machine entry is made possible by many PLD languages. PLD assemblers usually need not only Boolean literacy from system designers but also a detailed understanding of device architecture. State machine entry does not require such a high degree of understanding, so for these applications the logic designer can often use a descriptive rather than a mathematical language.

User-friendly software tends to make an architecture transparent, and state machine entry ensures rapid turnaround. Small PLD designs can be turned around in minutes, and more complex devices in several hours.

Users are also demanding more application-specific PLDs. At first appearing to be redundant, application-specific PLDs are devices whose architecture is aimed at a particular application.

The software of tomorrow will be more responsive to users' needs, allowing for a multitude of design descrip-

tions while not limiting the implementation to a particular device. Along with device independence will come the ability to handle multiple devices, with workstations offering the highest level of support.

Programming equipment must also do more than just generate waveforms. Advanced programmers will provide many test features, including complete functional testing. Or, programmers will be linked to more sophisticated testers.

PLD suppliers are increasing the quality of their products, while adding test features to verify that quality. Power-up, reset and register preloading simplify testing, but the programmer must be able to take advantage of them.

Just as today's PLD market is driven by manufacturers, tomorrow's will be guided more by the needs of the logic designer. As new applications arise, pressure will be put on both technology and architecture. Communication with the customer will assume new importance, since it defines the right combinations. In addition, software from third-party vendors will make programmable logic more at home across the board.

Just as today's market is driven by manufacturers, tomorrow's will be guided by designers. As needs arise, pressure will be on technology and architecture.

PLDs will continue to be used alongside discrete logic, fixed LSI functions, and gate arrays. But as the market matures, the benefits of the PLD over other options will become greater. In time, programmable logic will continue to enjoy a higher growth rate than the overall IC industry. ■

2

Introduction to Programmable Array Logic

A look at the architectural differences between PALs and other programmable logic devices

Vincent J. Coli

PROGRAMMABLE LOGIC devices are integrated circuits that hardware designers can program to perform specific logic functions. Most PLD functions are available from many vendors and in several technologies with different speed, power, and cost options. As with standard 7400 chips, PLDs are available off your local distributor's shelf. PLDs offer one distinct advantage over standard 7400 discrete logic: They are user-programmable.

Most PLDs consist of two arrays of logic gates—an AND array followed by an OR array. The input signals to a PLD must first pass through an array of AND gates where combinations of the input signals are formed. Each group of AND combinations is called a minterm in Boolean algebra or a product line in PLD nomenclature. Then the product lines are summed in an array of OR gates. The input buffers generate both the true and complement of the input signals.

Three basic types of AND/OR array-based PLDs exist: programmable read-only memories (PROMs), programmable logic arrays (PLAs), and programmable array logic (PAL) devices. The types are distinguished by the programmability of their arrays.

In a PROM, the AND array is fixed and the OR array is programmable. In a PLA, both arrays are programmable. PAL devices have a programmable AND array and a fixed OR array. I will compare the PAL device to the PLA and PROM and then examine the architecture of some commonly used PAL devices. For a brief history of the PAL device, see the text box "Evolution of PALs" by John Martin Birkner on page 208.

PROMs

While most people think of PROMs as devices for storing fixed programs and memory, the PROM is also ideal for logic applications requiring less than 10 inputs—especially when many product lines are required. PROMs designed as logic devices are usually referred to as PLEs (programmable logic elements).

Figure 1 shows the PROM's fixed-AND/programmable-OR arrays. For a discussion of notation used to describe PLD devices, see the text box "PLD Notation Panel" on page 210. Every input combination is available in the AND array, whether that combination of inputs is required or not. Since the AND array is hard-wired, it is not possible to perform logic minimization between input combinations.

The OR array is programmed to select the AND gate combinations (or product lines). Since every OR gate is connected to each product line, outputs may share product lines. For those familiar with memory design, the fixed-AND array is often called the address decoder, while the programmable-OR array stores the memory bits. Another way of looking at this is that PROMs store the logic transfer function as a lookup table in memory.

The advantage of PROMs is that every input combination can be decoded. The disadvantage is that the number of input pins available is restricted because the array size must be doubled for each additional input. The arithmetic works like this: A PROM with n inputs and m outputs requires an OR array of 2^n lines deep by m lines wide. For example, a PROM with 10 inputs and 8 outputs requires an OR array of 2^{10} by 8 or 8192 fuse locations. An 11th input would require that the array size be doubled to 16,384. Cost and performance constraints limit PROMs to 13 inputs and 8 outputs. PROMs designed specifically for logic applications feature either 5 or 6 inputs and 16 outputs.

PLAs

The PLA structure offers the highest level of flexibility because both arrays are programmable. Figure 2 shows the PLA's programmable-AND/programmable-OR structure. Because their OR arrays are programmable, PLAs, like PROMs, can share product terms among outputs. For example, one product line would be saved if two outputs required the same input combination (i.e., product line).

Programmability in the AND array removes the restriction found in PROMs that the AND array must be large enough to provide all possible input combinations. This works because, statistically, only a

continued

Vincent J. Coli is a strategic marketing manager for Monolithic Memories Inc. (2175 Mission College Blvd., Santa Clara, CA 95054). He has worked with the PAL products for the past six years with MMI. Vincent holds a B.S. in chemical engineering and an M.S. in electrical engineering.

Reprinted with permission from the January 1987 issue of BYTE magazine.
Copyright © by McGraw-Hill, Inc., New York 10020. All rights reserved.

The Evolution of PALs

John Martin Birkner

Computers used to be constructed from SSI, MSI, PROM, and RAM chips connected in jigsaw-puzzle fashion on many printed circuit boards plugged into a connector backplane. The computer designer's task was to build a functional unit such as a processor, disk controller, I/O controller, or memory board. If the design overflowed onto another board, connectors and ribbon cables had to be added. This made the design more expensive and sometimes risky due to noise coupling. The name of the game was to get it all on one board.

Mixing and Matching TTL Chips

Designers who had studied switching theory, information theory, Boolean algebra, and Quine-McCluskey minimization at college soon found that their textbooks would not be of much use. They learned that the practical art of computer design did not consist of optimizing an architecture with an orthogonal instruction set. It consisted of mixing and matching the collage of existing TTL chips onto a single board until an approximation of the design goal was reached. They did not design state-control sequence logic from top-down state-graph theory, but rather, slapped down a 74174 hexadecimal register and some 7400 NAND gates. Control-logic design theories usually consisted of following signal lines around the logic schematic until, through superhuman powers of concentration, designers achieved clarity.

Designers found the information they needed in the catalogs of young semiconductor companies in California, Arizona, and Texas. A favorite was *The TTL Databook* by Texas Instruments. Most logic designers believed that 74-series TTL parts found in this book would be second-sourced and could be "designed in."

A processor design would begin with the block diagram consisting of an ALU, data path and register file, microprogram memory and sequencer, and then a small and obscure block called "control logic." It might have been a small block on the diagram, but the control logic usually represented the majority of the chip count.

The control logic consisted of SSI/MSI gates and flip-flops connected together in random fashion, and there seemed to be no way to reduce it. The control logic also represented the area of highest design errors and was easily recognizable on the printed circuit board as the area with all the "cuts and jumpers." The engineering change notice (ECN) was the standard remedy for such errors and was a constant source of agony between manufacturing and engineering. Manufacturing would use yellow wires to stand out on the green PC board. Engineering would use green jumper wires to camouflage embarrassing mistakes.

The engineering manager would "pilot release" the current revision PC board as soon as the green wire count was low enough to pacify manufacturing. The design engineers would then flee to the next design, where they were expected to cram even more functions onto the single PC board to beat the competition's new threat.

I was convinced that there must be a better way to build computers. So, in 1975, I packed my bags and headed for Silicon Valley. I remember seeing the first single-chip microprocessor systems on the market. They had one microcomputer chip surrounded by a sea of over 100 SSI/MSI chips. The new LSI chips needed either some good planning so that they could talk to each other or some good "glue chips" to hook them up.

The PROM, pioneered by Harris and Monolithic Memories, showed some promise as a universal and general-purpose glue logic element. Applications like memory-address decoding began showing up for the 32-word by 8-bit PROM. National Semiconductor pioneered the programmable logic array (PLA) in a 14-in, 8-out, 96-product-term, 24-pin fat (0.6 inch wide) DIP, benchmarked for 96-character EBCDIC-to-ASCII conversion.

Intersil made a field programmable logic array, or FPLA, in the National pinout, but with about half the product terms at 48. Signetics increased the package pins to 28, making the 16-in, 8-out, 48-product-term 82S100 FPLA. These first attempts at providing the computer designer with LSI glue were met with mild enthusiasm. The new glue chips were too big (fat DIPs) and were slow, expensive, and hard to use.

Monolithic Memories was the first company to take advantage of the bipolar fuse-link PROM technology to make some fast little FPLAs, as we first called them. We put them in industry-standard 20-pin skinny (0.3 inch) DIPs, for minimum PC board area. We also reduced the two programmable arrays down to one for 35-ns high-speed operation and lower cost. We mimicked the TTL data-sheet specs down to the same terminology, graphics, and printing style to make the computer design engineer secure in replacing old 74-series TTL chips. We added programmable three-state output enable for I/O pin allocation. We added output registers with feedback for direct implementation of state-machine control logic from state graphs.

We designed the programming algorithm to be compatible with existing PROM programmers, making low-cost programming possible. The first PAL programming module had a PAL in it. This presented a chicken-and-egg problem that we solved by emulating with some PROM and SSI chips. The first PAL to be programmed was, of course, the pattern for the PAL programmer module. I headed the project, specified the design, and sold the customers. H. T. Chua provided a clever and ingenious circuit design.

New Design Methodology

The new chips required a new design methodology. Actually, it was the same method that we learned in school, so we had to drag out our old textbooks and relearn Boolean logic and top-down state-machine design. We showed the designer how to "design your own chip" using Boolean logic equations. We wrote the first silicon compiler, PALASM (PAL Assembler), and published the FORTRAN source in the *PAL Handbook* (available from McGraw-Hill), along with numerous design examples.

The PAL chips replaced SSI/MSI chips at a chip-count reduction of 5 to 1. Data General gambled on the new single-sourced chips by designing them into the MV8000 computer (see *The Soul of a New Machine* by Tracy Kidder).

Apple put six PAL chips in the Macintosh. Soon the PAL chips were no longer single-sourced, as National Semiconductor, Texas Instruments, Advanced Micro Devices, and others joined in licensing the now-patented PAL chips from MMI. Now you can find these chips everywhere. Look at the PC expansion boards in this magazine; you can recognize PAL chips by their easy-to-read part-number system (e.g., PAL16L8 and PAL16R8).

John Martin Birkner is president of Structured Design Inc. and coinventor of the programmable array logic (PAL) device. He can be contacted at 988 Bryant Way, Sunnyvale, CA 94087.

PLD Notation Panel

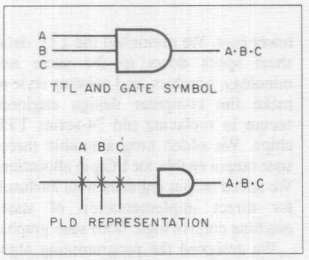


Figure A: *Differences in the logic notation for a TTL AND gate and a PLD AND gate.*

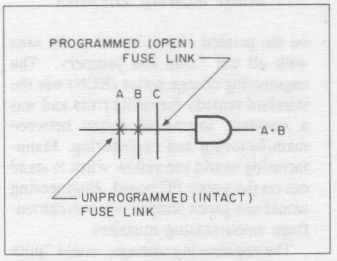


Figure B: *The partially programmed product line to implement A*B.*

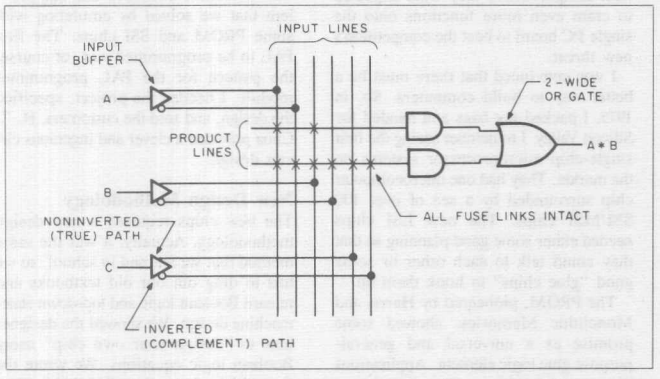


Figure C: *Portion of the PLD array programmed to implement A*B. Having all fuses intact in the second product line causes a logic zero to be input to the second AND gate, which does not contribute to the sum at the OR gate.*

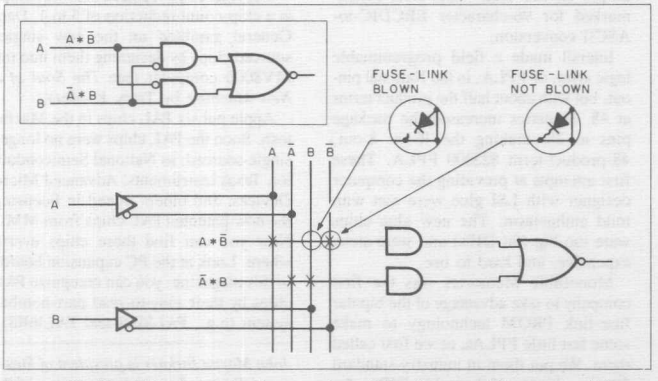


Figure D: *The PAL device implementation of the function /Output = A*B + /A*B. The standard combinatorial logic diagram of the function /Output = A*/B + /A*B is shown in the inset to figure D above.*

Because PLD structures are much different from ordinary TTL gates, new logic notations were developed for them. Figure A shows the logic convention adopted for a three-input AND gate. The PLD representation for an AND gate is called a "product line." Note that the three vertical lines are the inputs (A, B, and C), which are connected to the AND gate inputs through fuse links. An unprogrammed (or closed) fuse link is represented by an X at the intersection of an input line with a product line. If you wanted to disconnect one of those inputs from AND gate C, for example, you would remove the appropriate X from the point of intersection for the C input line with the product line to signify a programmed (or open) fuse link. This product line, which now implements the A*B function, is shown in figure B.

Since every input is available to every product line in a PLD, it is convenient to show the input lines as long lines running vertically through the array. Also, two input lines are associated with each input pin because both input polarities are available in a PLD. Therefore, the input buffer is shown with both a noninverted (true) and inverted (complement) output path; each path is hardwire connected (shown as a dot) to an input line.

Figure C shows a portion of a PLD array illustrating the input lines and buffers. Notice that an OR gate is added to the structure. All the fuse links in the lower product line are left intact, leaving the product line in a logic low (since true inputs are ANDed with complements), while appropriate fuse links in the upper product line are programmed to implement the A*B function from figure B.

It is common to implement two or more levels of logic gates such as an AND/OR/invert circuit in a PLD. For example, consider the following function implemented in a PAL device:

/Output = A*/B + /A*B.

Shown in figure D are the standard combinatorial logic diagram (see inset) and the PAL logic equivalent for this function.

Notice the details added to figure D that magnify the programmed fuse link for B in the upper product line. This magnification details each fuse link and its associated diode for a bipolar PAL device. A CMOS PAL device is similar, except an ultraviolet cell would substitute for the fuse link.

limited number of product terms is required in any equation. Eliminating redundant combinations with logic minimization techniques, such as Karnaugh maps, can reduce the required number of product terms even more. Therefore, almost any combination of inputs can be decoded in a PLA.

PLAs were the first products offered specifically for logic applications. Due to programming limitations, early PLAs were available only in mask-programmed versions. Just like on a ROM, a logic designer would indicate on the vendor's PLA AND/OR logic map where the desired connections were to be made. The vendor would then tool up a custom metal mask for the PLA to implement the customer's logic. Today, most PLAs are user-programmable. However, mask-programmed PLA structures are used often in the control section of LSI/VLSI standard logic chips, such as microprocessors, and offered in standard-cell libraries.

In the world of engineering, there are always compromises. The facts reveal that a performance and silicon-die size penalty must be realized to provide the flexibility of programming both arrays. PAL devices are generally 5 to 10 nanoseconds faster than PLAs at the same power level and save the silicon area required to program and verify the second array. It turns out that the flexibility of a programmable-OR array is not required for most PLD applications, but it can be useful for complex state-machine and sequencer applications.

Because of the long history of PLAs, their nomenclature can be a little confusing. Early vendors of user-programmable PLAs called their products FPLAs to highlight their "field programmability" and to distinguish FPLAs from factory mask-programmed PLAs. Just as ROMs and PROMs could be easily distinguished, so could PLAs and FPLAs. However, since most of the PLAs offered today are programmed by the customer, many vendors have dropped the F prefix and simply call them PLAs. Furthermore, PLAs designed for sequencer applications are called PLSs (programmable logic sequencers).

PAL Devices

Figure 3 shows the programmable-AND/fixed-OR array structure of a PAL. As with the PLA, having the AND array programmable lets the user program only the desired input combinations. But fixing the OR array requires that certain product lines be tied to specific outputs—typically, eight product lines per output.

Many people use PAL and PLD synonymously. Several PLD vendors add an E prefix to PLD, to come up with EPLD, which signifies ultraviolet erasable PLDs. Just as there are PROMs and EPROMs, now there are PLDs and EPLDs.

The name HAL (hard array logic), refers to mask-programmed, or ROM, versions of PAL devices. If the volume of devices needed were large, converting a design to a HAL might be appropriate once the design is thoroughly debugged with the PAL.

While all PAL devices are characterized by a programmable-AND/fixed-OR array structure, there is a whole line of PAL devices with different options. They come with varying numbers of inputs and outputs. They might have feedback paths from the output back to the array. Some of these pins can be programmable I/O pins. They can have active-high or active-low outputs, or the output polarity might be programmable via an XOR gate and a fuse. Some come with registers at their outputs and are good for making sequential circuits. Let's look at two commonly used PAL devices: the 16L8 and the 16R8.

The PAL16L8

One popular combinatorial PAL is the PAL16L8 (figure 4). Notice how the pins on the left side and bottom of the logic diagram (pins 1 to 9 and pin 11) are used for inputs and the pins on the right (pins 12 to 19) are available as outputs. Pins 12 and 19 can be used only as outputs, but six of the outputs (pins 13 to 18) are also

continued

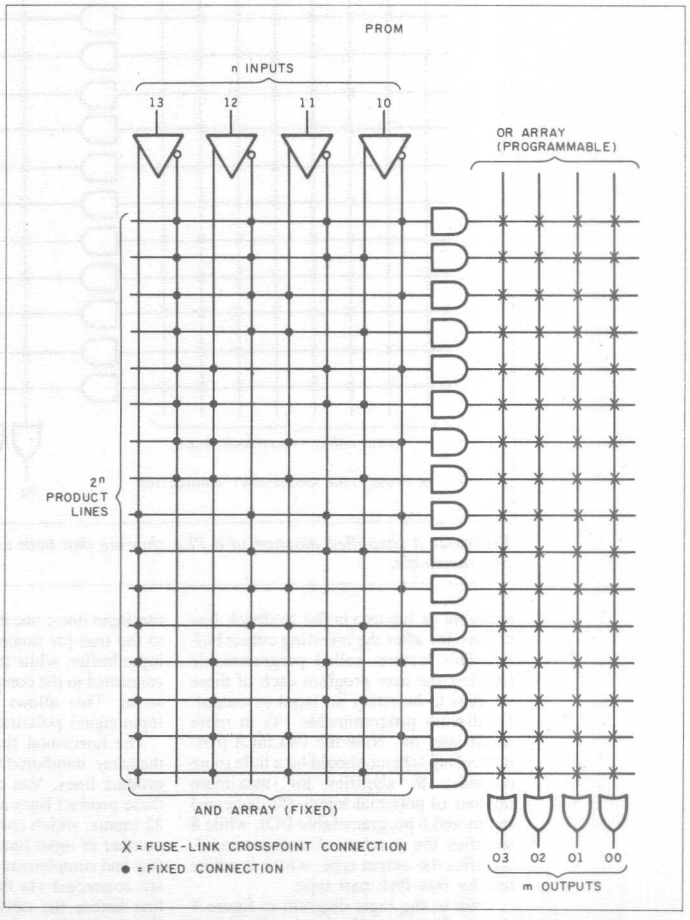


Figure 1: *A simplified diagram of the fixed-AND/programmable-OR array structure of the PROM. Figures in this article are reprinted from Monolithic Memories'* PAL Handbook *(3rd ed.) with permission from Monolithic Memories.*

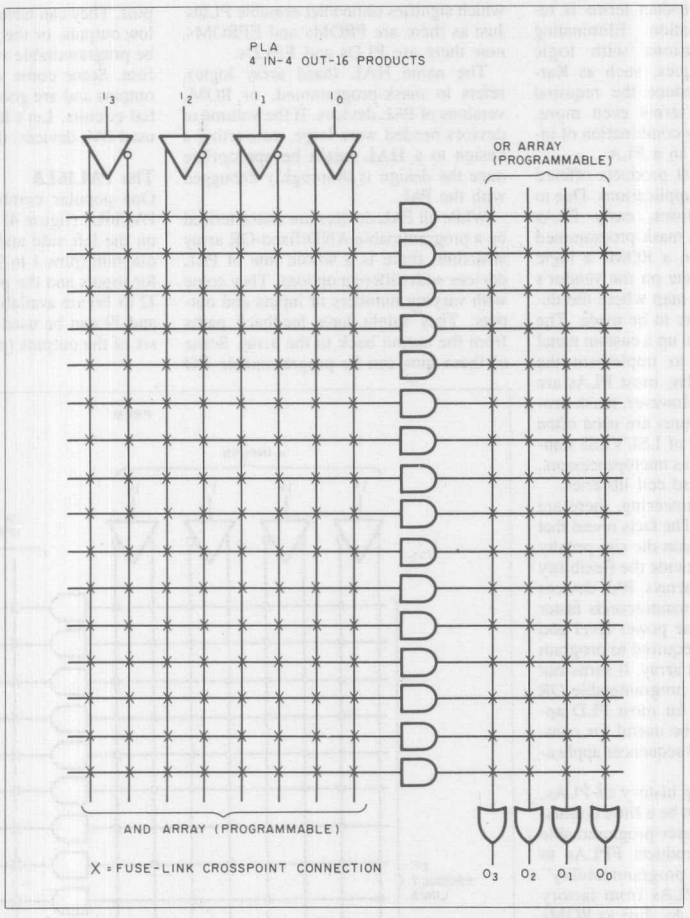


Figure 2: *A simplified diagram of a PLA showing that both arrays are programmable.*

available as inputs via the feedback line connection after the inverting output buffer. This feature, called programmable I/O, lets the user program each of these six pins to be either an input or output. I'll discuss programmable I/O in more detail later on. Now the PAL16L8 part-numbering scheme should be a little more obvious; 16 signifies the maximum number of potential inputs (10 dedicated inputs and 6 programmable I/O), while 8 signifies the number of outputs and L signifies the output type, which is active low for this PAL part type.

Refer to the logic diagram in figure 4 and you'll see that the vertical lines running through the array, numbered 0 through 31, are the input lines. Notice that each input or I/O pin is associated with two input lines; one input line is connected to the true (or noninverted) sense of the input buffer, while the other input line is connected to the complement (or inverted) sense. This allows availability of both input-signal polarities to the array.

The horizontal lines running through the array, numbered 0 through 63, are the product lines. You can think of each of these product lines as an AND gate with 32 inputs, which corresponds to the total number of input lines. Actually, both the true and complement of every input signal are connected via fuses to each product line before the device is programmed. This is the programmable-AND array in the PAL structure. To program the array, the user selects different combinations of

continued

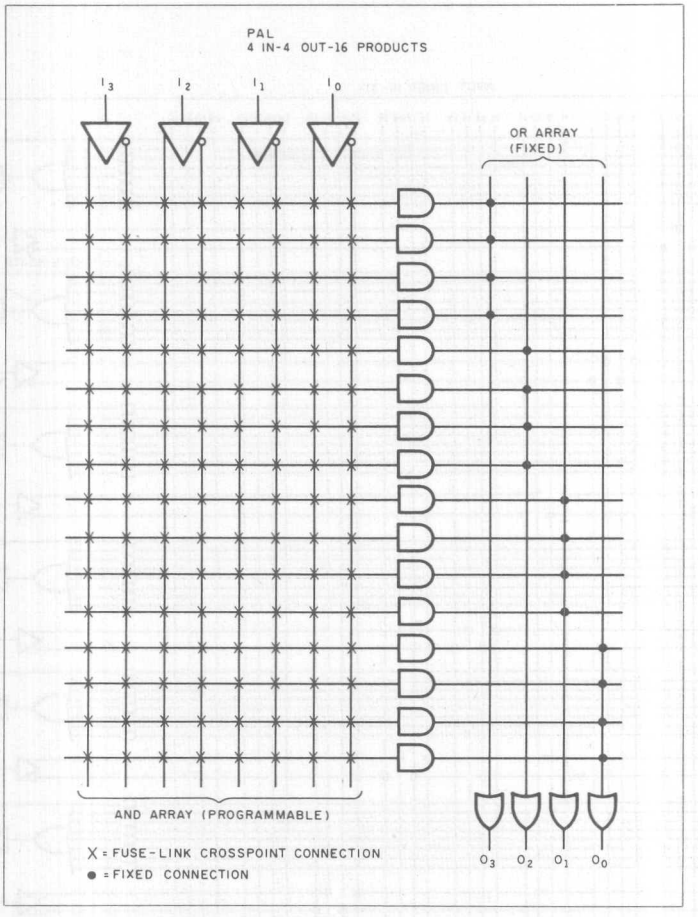


Figure 3: *A simplified diagram showing the programmable-AND/fixed-OR array structure of a PAL device.*

input signals by disconnecting, via the blown fuse, the unwanted input signals in a product line. In total, 2048 fuses are available in this PAL device (64 product lines by 32 input lines).

Notice that each output pin has eight product lines associated with it. The lower seven product lines of each group are summed at the OR gate, while the upper product line is connected to the inverting output buffer. The lower seven product lines and the OR gate provide the sum-of-products logic power for the PAL device. The OR gate determines whether any of the product lines are active, or true, and then the output buffer inverts the signal from the OR gate for output. Note that a product line with all fuses left intact will not affect the sum at the OR gate, since the logical result of each input ANDed with its complement is false.

Programmable I/O

This upper product line associated with each output controls the three-state logic in the output buffer. When this product line is active, or true, the output is enabled and the sum-of-products logic determines the output state. However, when this product line is inactive, or false, the output is disabled with the three-state buffer in the high-impedance state. This lets the output pin drive a three-state bus just like a 74S240 octal buffer. Furthermore, since most PAL devices feature an output drive capability of 24 milliamperes, they are quite handy for bus interfacing.

continued

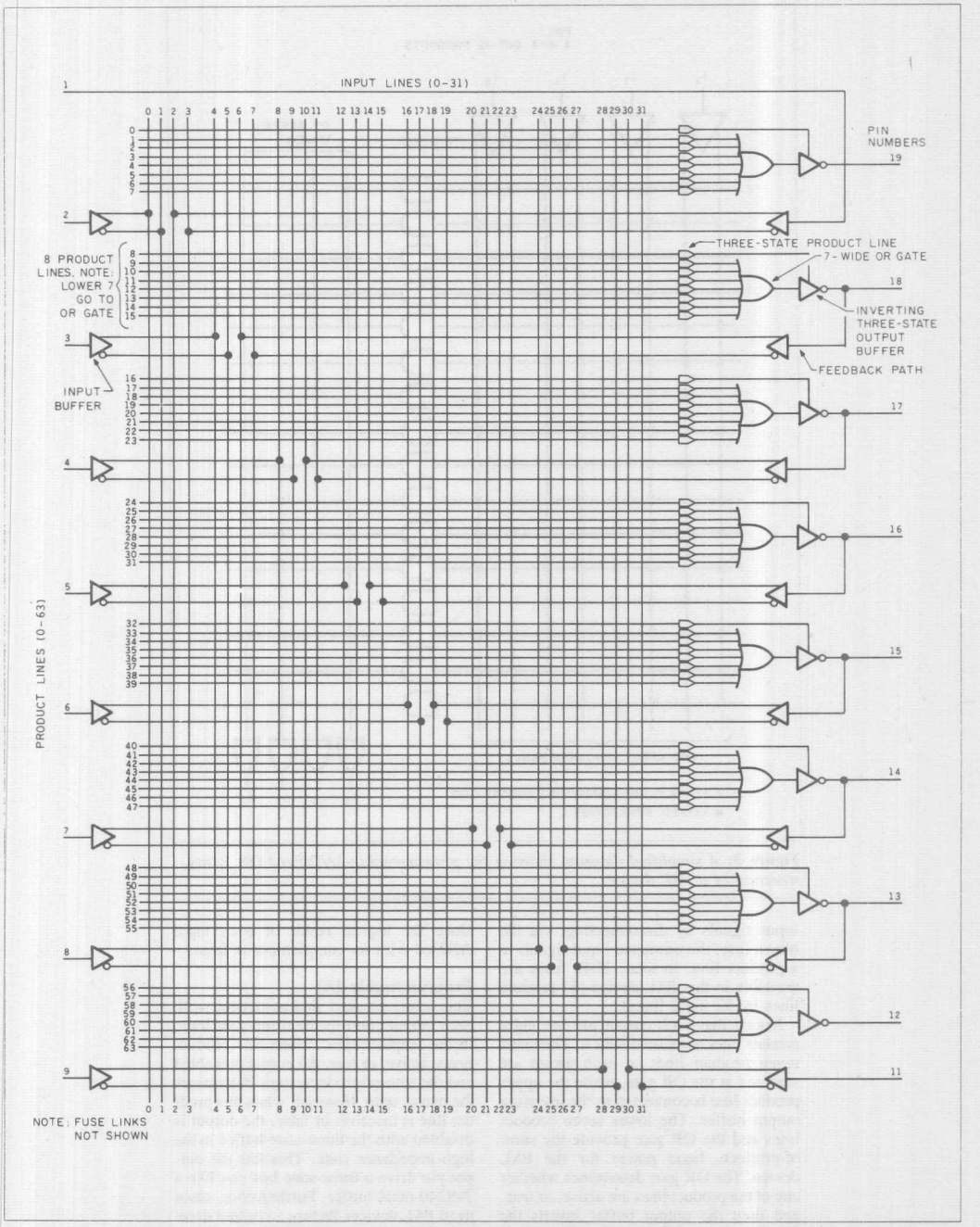


Figure 4: *The actual logic diagram of the combinatorial PALI6L8. The fuse links are not shown so that a logic designer may place an X at those points where a fuse should be left intact.*

The three-state product line, along with the feedback path on six of the outputs, makes the programmable I/O feature work. The pin is an input to the AND array when all the fuses in the three-state enable product line are left intact, while the pin is an output when all the fuses are programmed. Note that a product line will always be true, regardless of input combinations, when all fuses are programmed. The programmable I/O feature lets the user allocate pins for input or output as required by the application.

An even higher level of flexibility is possible if you let the logic in the product

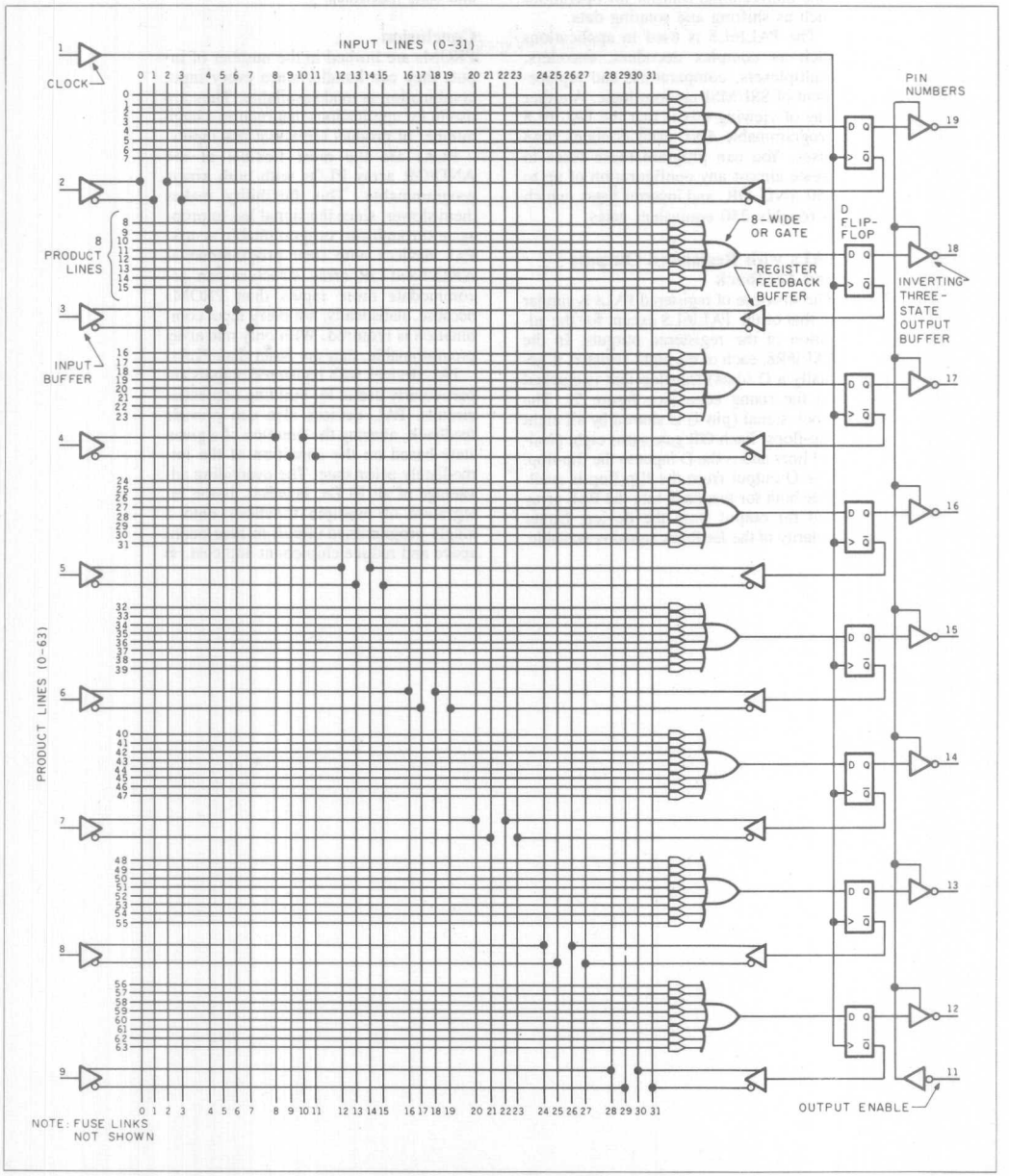


Figure 5: *The actual logic diagram of the PALJ6R8. Notice the D-type flip-flops at the outputs.*

line determine the pin's direction. This is done by programming a condition in the product line for which the pin will be an output. You can use this feature to allocate available pins for I/O functions or to provide bidirectional transfer for operations such as shifting and rotating data.

The PAL16L8 is used in applications such as complex decoders, encoders, multiplexers, comparators, and replacement of SSI/MSI random logic. Another way of viewing this is that the PAL16L8 programmable AND array contains 2048 fuses. You can program these fuses to create almost any configuration of up to 250 AND, OR, and inverter gates, which is roughly 250 equivalent gates.

PALs with Registered Outputs and Feedback

The structure of registered PALs is similar to that of the PAL16L8 except for the addition of the registered outputs. In the PAL16R8, each of the eight registers is actually a D (data) flip-flop that is clocked on the rising edge (see figure 5). The clock signal (pin 1) is shared by all eight flip-flops. Each OR gate sums eight product lines and is the D input to the flip-flop. The Q output from the flip-flop is available both for feedback into the PAL array and for output from the device. Either polarity of the feedback signal is available.

This feedback lets the PAL device "remember" the previous state, and it can alter its function based upon that state. Thus, registered PAL devices are ideal for implementing single-chip state sequencers and state machines.

Conclusion

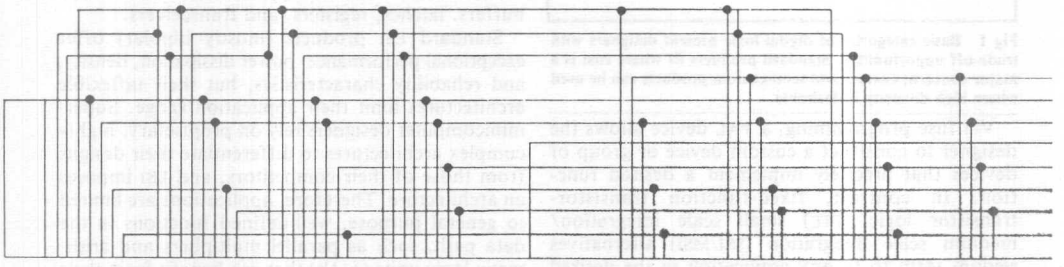
PROMs are limited in the number of inputs they can handle, since every input combination is made available. They are useful for applications that require a large number of product terms but few inputs.

PLAs are the most flexible of the AND/OR array PLDs with both arrays programmable. This flexibility makes them slower, since the signal has to propagate through two programmable arrays. PAL devices with their programmable-AND/fixed-OR array structure can accommodate more inputs than PROMs because, statistically, not every input combination is required. With only one array programmable, they are faster than PLAs.

PAL devices with registered outputs are particularly useful for building sequential circuits. PAL devices also can provide feedback, altering the function of a given state based on the condition of the immediately prior state. The overriding advantage of all PLDs, however, is the integration of multiple functions onto a single programmed circuit to save board space and reduce chip count and cost. ■

Logical Alternatives in Supermini Design

Becoming the technology of choice in superminicomputer design, programmable array logic mixes with alternative devices to meet design constraints.



by Bradford S. Kitson and B. Joshua Rosen

Programmable array logic devices have been a driving force behind the latest generation of 32-bit superminicomputer designs. These superminicomputers range from redesigns of existing architectures that reduce cost or overcome packaging limitations, to designs requiring ultrafast turnaround, to high performance architectures specified to take full advantage of such devices.

Superminicomputers designed with programmable array logic (PAL®) include the VAX®-11/730 from Digital Equipment Corp, the MV/8000 and MV/10000 from Data General, and Computervision's APU™ (analytic processing unit). A reimplementation of the original VAX-11/780, the VAX-11/730 supplies 25% of the performance in 10% of the board space. Programmable array logic was chosen by the MV/8000 designers to allow the shortest possible design cycle and to catch up with their competitors. The MV/10000 upgrades performance of the MV/8000. The Computervision APU was designed for high performance with PAL devices in mind, and provides excellent examples of how to use such devices to their fullest.

Bradford S. Kitson is section manager in product planning and applications for programmable logic devices at Advanced Micro Devices, 901 Thompson Pl, Sunnyvale, CA 94088. He holds a BS in electrical engineering and computer science from the University of California at Berkeley.

B. Joshua Rosen is manager of processor design at Dataflow Systems Corp, 42 Nagog Park, Acton, MA 01720. He was manager of processor development at Computervision Corp, Bedford, MA, when this article was written. He holds a BA in physics from Lawrence University and an MSEE from Northwestern University.

Logic Design Alternatives

Although by no means the only option, programmable array logic has become the technology of choice in superminicomputer design. Logic design alternatives (Fig 1) include standard products (fixed-function devices), semi-custom (programmable logic, gate arrays, and standard cells), and fully-custom logic devices. In superminicomputer design, the primary alternatives are standard products, gate arrays, and PAL devices. The best choice for any given function depends upon the design alternative capabilities, the design constraints, and the function actually being implemented.

Reprinted with permission of COMPUTER DESIGN

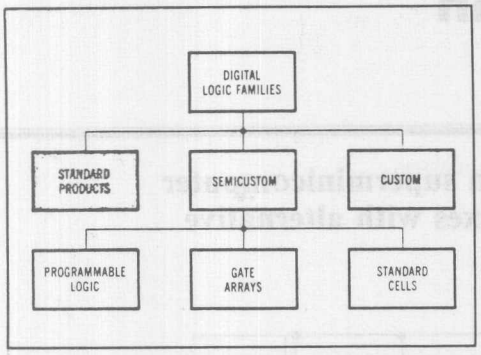


Fig 1 Basic categories of digital logic present designers with trade-off opportunities. Standard products fit where cost is a major concern; custom and semi-custom products can be used where high diversity is desirable.

Via fuse programming, a PAL device allows the designer to construct a custom device or group of devices that precisely implement a desired function. In contrast, fixed-function transistor-transistor logic (TTL) small scale integration/medium scale integration (SSI/MSI) alternatives seldom seem to fit any application in the desired way. Thus, the SSI/MSI designer usually pays penalties via extra logic levels in the critical path and an increased package count. Gate arrays allow custom devices to be created via mask programming. However, designers using a gate array must finalize architecture early in the design cycle. Any errors will require a mask change, which can take months.

In most cases, the best choice is a combination of the alternatives and probably includes some memory as well. Typically, the six basic design constraints are performance (speed), cost, density (packaging), power dissipation, reliability, and design turnaround. Assigning a priority to these constraints will usually define the logic alternative that a machine is based on. Actual implementation trade-offs are made at the function level. The three basic functional portions of a design are data path, control path, and interface.

Standard Products Have Their Place

Standard products are defined as devices created for a wide market. Examples of standard products are TTL SSI/MSI, fixed instruction set metal oxide semiconductor (MOS) microprocessors, and microprogrammable large scale integration (LSI) building blocks. These devices are usually multiple sourced and produced in high volume, resulting in lower individual device costs. In a design where cost is the main concern, and performance, power dissipation, density, and design turnaround are of little or no importance, standard products are probably the best choice. In a design such as a superminicomputer, where these other considerations have a high priority, inherent disadvantages limit standard product use.

While SSI/MSI devices offer fast individual gates, on a system level their density, power dissipation, and reliability characteristics are not as good as those of the alternatives. In addition, design turnaround characteristics are inadequate because changes usually require printed circuit (PC) boards to be laid out again. In most superminicomputers, therefore, SSI/MSI gates are used only in selected critical path functions that require one or two gate levels (at the expense of density and power dissipation), and in interface applications, such as bus buffers, latches, registers, and transceivers.

Standard LSI products (mostly bipolar) offer exceptional performance, power dissipation, density, and reliability characteristics, but their inflexible architectures limit their application range. Superminicomputer designers rely on proprietary, highly complex architectures to differentiate their designs from those of their competitors, and LSI imposes an architecture. Therefore, applications are limited to general purpose, well-defined functions in the data path, such as parallel multipliers and arithmetic logic units (ALUs) that can benefit from their high performance characteristics.

Gate Arrays Are "Cast in Concrete"

Semi-custom gate arrays are defined by integrated circuit (IC) manufacturers as large arrays of unconnected gates. End users specify how gates are interconnected with actual interconnection occurring at the metal-mask layer of the IC process. The main advantages of gate arrays are high density and the ability to customize a design, while primary disadvantages are cost and design turnaround. Each custom device is single sourced and low volume; therefore they are not cost-effective unless the application is density limited or the volume is very high. Gate arrays can adversely affect design turnaround because the system designer must design both the IC and the system in which the IC is used. In addition, any change in the gate array requires new masks and a delay for each mask iteration.

By using gate arrays only where the design can be defined early, designers minimize these turnaround disadvantages. They then hedge their bets by surrounding the gate arrays with logic that can correct any design bug(s) discovered later on. As in LSI, what is "cast in concrete" is usually the data path. However, gate arrays allow more of the data path to be integrated because the device can be optimized for specific design requirements. A proprietary 16-bit ALU slice might be a typical gate array.

Gate arrays are also used to interface multiple onboard buses together as data path "glue." Control-path and interface applications, however, tend to be too likely to change. Therefore, control

path functions are based on implementation techniques such as writable-control-store (WCS). Interface applications frequently require changes because of the need to interface one designer's board to another's. Consider the critical timing between a cache, an instruction fetching unit, and multiple register-files. The exception handling capability required for typical operations, such as a cache miss, can be an indeterminant problem affecting all of the above-mentioned functional units in the system. Should a bug occur in the cache unit, the interface section of all other units will have to be changed to accommodate the correction.

PAL Structure Paves the Data Path

Combining simplicity and flexibility, the basic PAL structure is a fuse-programmable AND gate array that drives fixed connection OR gates, allowing logic to be implemented in sum-of-products (AND-OR) Boolean form.

By selectively blowing the appropriate fuses, a PAL device can implement any logic function as long as the number of inputs or AND gates required does not exceed the number provided in the device chosen. In the AmPAL16R4, for example, the true and complement version of each of the 16 inputs is connected via fuses to each of the 64 AND gates in the device (Fig 2). PAL devices provide additional features, such as programmable input/output (I/O) pins and registered outputs with internal feedback that further enhance their ability to implement logic functions efficiently. Programmable I/Os are especially useful for trading off the number of device outputs for inputs to fit the exact number required by the logic functions being implemented. Internal registered feedback is desirable when implementing complex state machine designs.

Programmable logic devices, like gate array, are semi-custom devices. Created by the IC manufacturer, they are alterable by fuse programming for a specific application. Designed specifically for logic-oriented applications, PAL devices enable designers to create custom devices with fast turnaround time. PAL devices compare favorably to alternative devices in terms of performance, cost, density, power dissipation, and reliability. They fall behind gate arrays and LSI in density and power dissipation, and behind SSI/MSI in individual device cost.

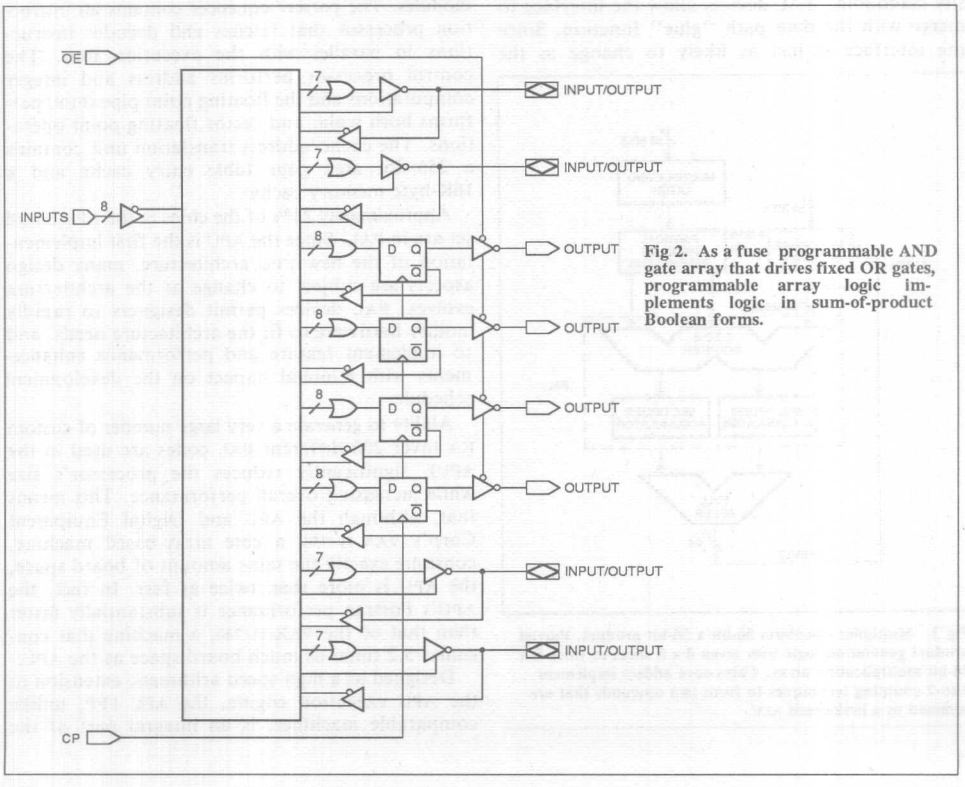


Fig 2. As a fuse programmable AND gate array that drives fixed OR gates, programmable array logic implements logic in sum-of-product Boolean forms.

(Note however, that PAL devices are cheaper on a system basis.)

Performance and density characteristics of PAL devices have led to significant applications in the data paths of superminis where they are used along with LSI and/or gate arrays. These devices serve to "glue" the LSI and the "cast in concrete" gate arrays into the system. If necessary, the design turn around capability that they display can be used to optimize the data path architecture and, in some cases, fix a bug in a gate array by reprogramming the PAL devices around it. Typical data path functions for the devices include barrel shifters, masking, code conversion, and multiple bus interface.

Design turnaround becomes especially beneficial in the control path and interface portions of a design. Most control path functions are highly random and are prone to change and/or error While WCS allows design changes to be made by rewriting microcode, PAL devices permit changes that cannot be made in microcode, or would adversely affect system performance. For instance, one superminicomputer manufacturer has established the rule that a gate array can be used in the control path only if eight or more PAL devices are necessary for the same function. In interface design, from a density standpoint, PAL devices allow the interface to merge with the data path "glue" function. Since the interface is just as likely to change as the control path, but without the benefit of WCS, design turnaround becomes a factor. If one board's designer needs to change the interface, the designs of many other boards are impacted. Updates on all boards can be easily made by reprogramming one or more PAL devices. These devices can also provide the drive capability required by this application area.

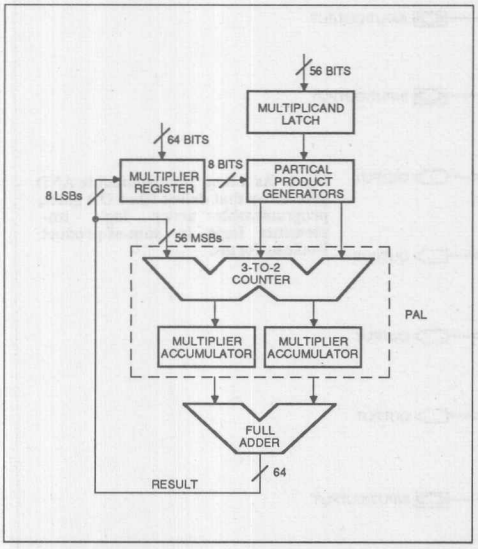


Fig 3 Multiplier calculates 56-bit x 56-bit product. Partial product generation logic uses seven 8 x 8 slices to form 8 x 56-bit multiplication array. Carry-save adders implement 3-to-2 counting techniques to form two operands that are summed in a lookahead ALU.

Designing with PAL Devices in Mind

Trade-offs among PAL devices and other digital logic alternatives were key factors in Computervision's APU design. In conjunction with standard products, such as ALUs, LSI multipliers, and memory devices, PAL devices were used to create a patented architecture not feasible in standard TTL SSI/MSI.

Designed as a very high speed 32-bit superminicomputer for engineering applications, the Computervision APU is twice as fast as competitive designs, yet occupies the same board space. Its designers, instead of merely replacing TTL SSI/MSI with PAL devices, used PAL as customizable logic building blocks. This allowed them to implement powerful logic functions in a minimum of space.

The APU processor board set is divided into four modules. The parser/sequencer contains an instruction processor that fetches and decodes instructions in parallel with the execution unit. The control processor performs address and integer computations and the floating point pipe (FPP) performs both scalar and vector floating point operations. The cache/address translation unit contains a 256-slot area page table entry cache and a 16K-byte memory cache.

Approximately 25% of the chips in the APU board set are in PAL. Since the APU is the first implementation of the new CPU architecture, many design aspects are subject to change as the architecture evolves. PAL devices permit designers to rapidly modify hardware to fit the architecture needs, and to implement feature and performance enhancements with minimal impact on the development schedule.

Ability to generate a very large number of custom ICs (over 200 different PAL codes are used in the APU), significantly reduces the processor's size while increasing overall performance. This means that, although the APU and Digital Equipment Corp's VAX-11/750, a gate array-based machine, consume exactly the same amount of board space, the APU is more than twice as fast. In fact, the APU's Fortran performance is substantially faster than that of the VAX-11/780, a machine that consumes 5.2 times as much board space as the APU.

Designed as a high speed arithmetic extension to the APU execution engine, the APU FPP, unlike comparable machines, is an integral part of the

internal architecture and not an optional add-on. As a result, the FPP not only accelerates scalar and vector floating point arithmetic, but also performs byte, word, double-word, and quad-word string operations. In addition, it serves to enhance the performance of important nonfloating point instructions such as Procedure Call and Return. A total of 79 PAL devices are used for both control- and data-path applications on the FPP board.

Counter Solves Multiplication Problem

The heart of the FPP is the multiplier section (Fig 3). Double-precision floating point multiplication requires the calculation of a 56-bit x 56-bit product. Unfortunately, 56 x 56 parallel multipliers do not exist on silicon. From a cost/performance standpoint, the best solution is to use a number of small multipliers to build an intermediate sized parallel multiplier and then produce a large product (56 x 56) in multiple cycles.

Partial product generator FPP logic uses seven Am25S558 8 x 8 multiplier slices to implement an 8 x 56-bit multiplication array. Each multiplier chip produces a 16-bit product. In general, the 8 most significant bits (MSBs) of each partial product generator must be added to the 8 least significant bits (LSBs) of the next higher slice to generate the full 64-bit partial product. Exceptions are the 8 MSBs and LSBs.

This technique also requires the ability to accumulate partial products with the partial products from previous cycles. Thus, each cycle must be accompanied by two additions: the partial product summation and the intermediate product accumulation. While this can be done by following the multipliers with two levels of lookahead adders, usually 74S181s, the resulting nanocycle time is approximately three times longer than the partial product generation time of the 8 x 8 multipliers. Modifying this scheme, however, by adding registers between each level of logic, the pipeline multiplier reduces the nanocycle time to near the propagation delay time of the multiplier chips plus the clock to output time of the multiplier register plus the setup time of the intermediate result register. This scheme has two disadvantages: increased pipe latency, caused by the two extra levels of pipelining, and a high parts count.

Still another technique involves replacing one level of the pipe and one level of lookahead adders with carry-save adders between the partial product generators and the pipeline registers. Carry-save adders are used to implement a technique that is called 3-to-2 counting. As seen in Fig 4(a), any

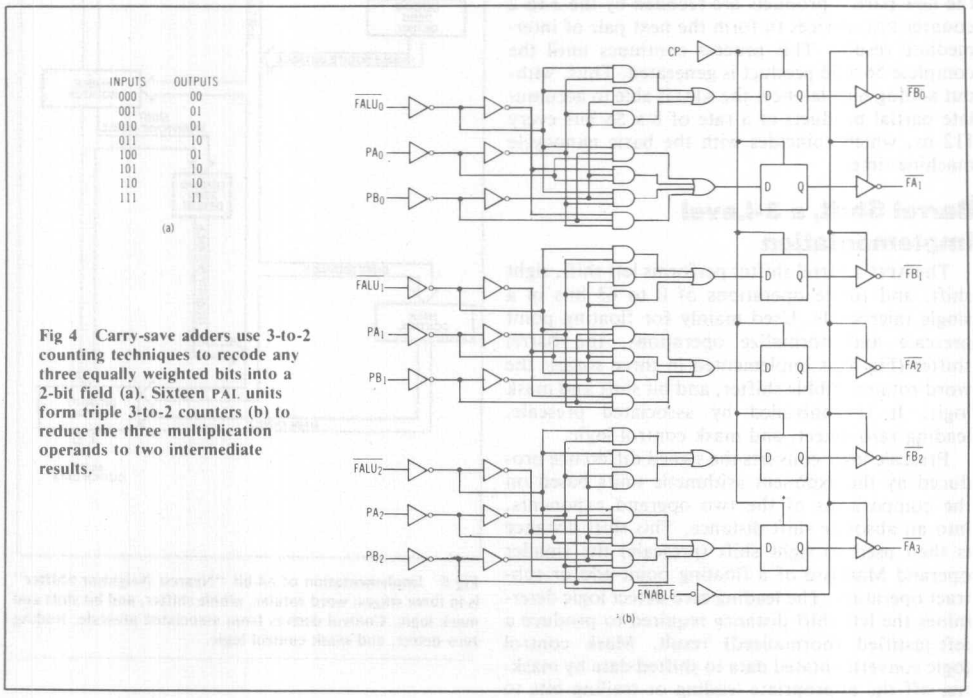


Fig 4 Carry-save adders use 3-to-2 counting techniques to recode any three equally weighted bits into a 2-bit field (a). Sixteen PAL units form triple 3-to-2 counters (b) to reduce the three multiplication operands to two intermediate results.

Without adding pipe latency, the APU is able to accumulate partial products at a rate of 8 x 56 bits every 112 ns.

combination of 3 equally weighted bits can be recoded into a 2-bit field.

This makes it possible to reduce the three operands generated by the multiplication process (high and low partial products and 64-bit intermediate product) into only two operands which may then be summed together in a single lookahead ALU. The 3-to-2 recoding requires no-carry propagate logic and is therefore very fast. Only one level of pipelining is required because of the 3-to-2 counter's speed, resulting in both a reduced parts count and a reduced pipe latency.

In the APU floating point engine, 16 AmPAL16R6s, programmed as triple 3-to-2 counters, are used to reduce the three multiplication operands to two intermediate results [Fig 4(b)]. The registered PAL outputs are connected to the input buses of the Mantissa ALU that is also used for floating point addition and subtraction. The Mantissa ALU then calculates the next intermediate product in parallel with the partial products calculations occurring in the 8 x 8 multipliers. This intermediate product and the new partial products are recoded by the 3-to-2 counter PAL devices to form the next pair of intermediate results. This process continues until the complete 56 x 56 product is generated. Thus, without adding pipe latency, the APU is able to accumulate partial products at a rate of 8 x 56 bits every 112 ns, which coincides with the basic nanocycle machine time.

Barrel Shift, a 3-Level Implementation

The APU's barrel shifter performs left shift, right shift, and rotate operations of 0 to 63 bits in a single microcycle. Used mainly for floating point prescale and normalize operations, the barrel shifter (Fig 5) is implemented in three stages: the word rotater, nibble shifter, and bit shift and mask logic. It is controlled by associated prescale, leading zero detect, and mask control logic.

Prescale logic converts the signed difference produced by the exponent arithmetic units based on the comparisons of the two operand exponents, into an absolute shift distance. This shift distance is then used to right shift (prescale) the smaller operand Mantissa of a floating point add or subtract operation. The leading zero detect logic determines the left shift distance required to produce a left-justified (normalized) result. Mask control logic converts rotated data to shifted data by masking off the appropriate leading or trailing bits to implement right or left shifts. These three sections are implemented in PAL.

Implementing the 3-level, 64-bit barrel shifter in MSI requires the use of Am25S10 4-bit shifters. The first level is the word rotater which performs a circular rotate of 0, 16, 32 or 48 bits. Although implementation is simple, the MSI solution requires 16 packages. The second level is the nibble shifter, essentially identical to the word rotater but wired to rotate 0, 4, 8, or 12 bits. The final barrel shifter stage requires not only bit rotate but also leading and trailing bit masking and sticky bit computation (ie, the logical OR of the masked-out bits). An MSI solution requires not only the 16 packages of AM25S10s, used in each preceding level, but also 16 AND gate packages for masking, with another 16 AND packages for the sticky bit computation. Control logic for the mask operation requires as much logic as the entire shift path. A more practical solution consists of building separate left and right

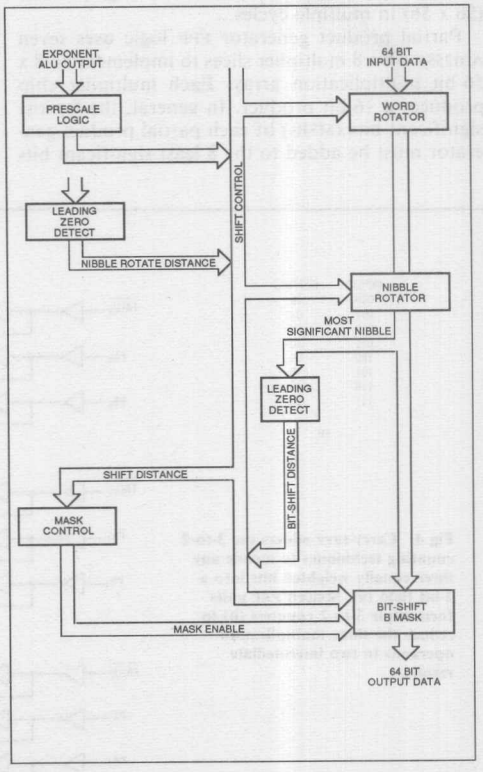


Fig 5 Implementation of 64-bit "Nearest Neighbor Shifter" is in three stages: word rotater, nibble shifter, and bit shift and mask logic. Control derives from associated prescale, leading zero detect, and mask control logic.

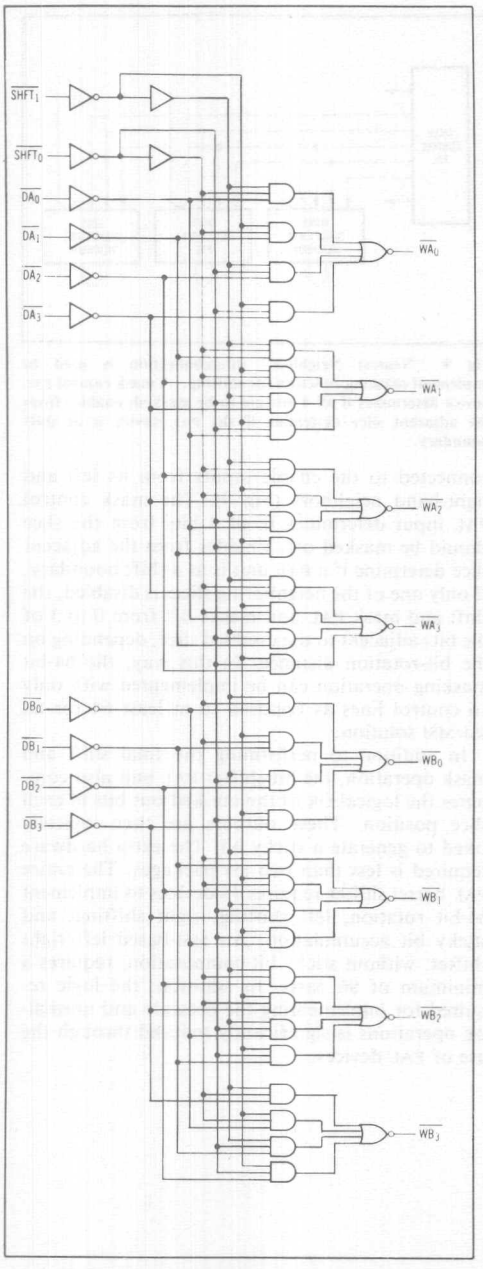


Fig 6 Word rotation, first level of 3-level shifter, consists of eight PAL units, each programmed as two 4-bit rotators. It performs a circular rotate of 0, 16, 32, or 48 bits.

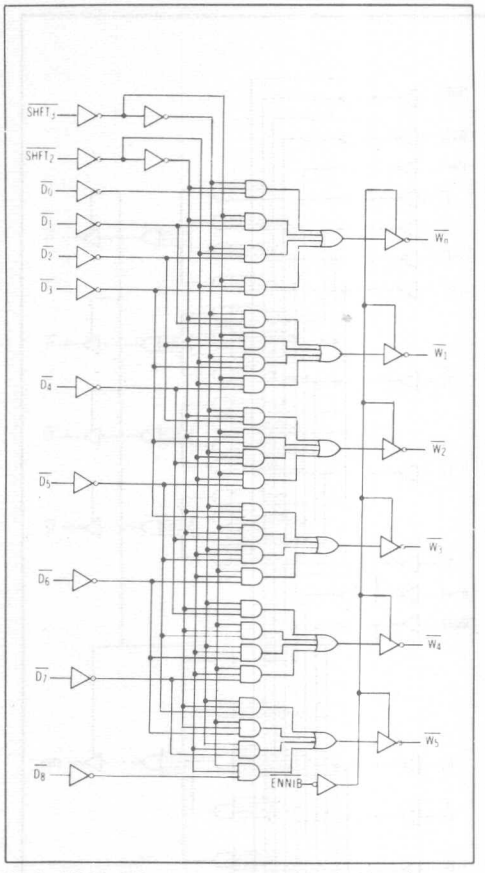


Fig 7 Nibble shifter makes up second level of barrel shifter. It is wired to rotate 0, 4, 8, or 12 bits.

shifters, 48 packages apiece, and not implementing a sticky bit at all.

Implemented in PAL, a 64-bit rotater and shifter with sticky bit computation requires considerably fewer packages than a unidirectional MSI shifter. The word rotater consists of eight identical PAL devices programmed as two 4-bit rotaters/package (Fig 6). The nibble shifter requires four Am25S10s and eight PAL devices programmed as 6-bit wide, 4-place shifters (Fig 7). The bit shift and mask logic requires 16 PAL devices in the data path and two PAL devices in the control path. (Fig 8).

To implement the masking function required for shifting, a technique called "Nearest Neighbor Shifting" (U.S. patent pending, Computervision Corp) is used. Each shift and mask PAL device has an enable input from one mask control PAL unit. In addition, each shift and mask PAL device is also

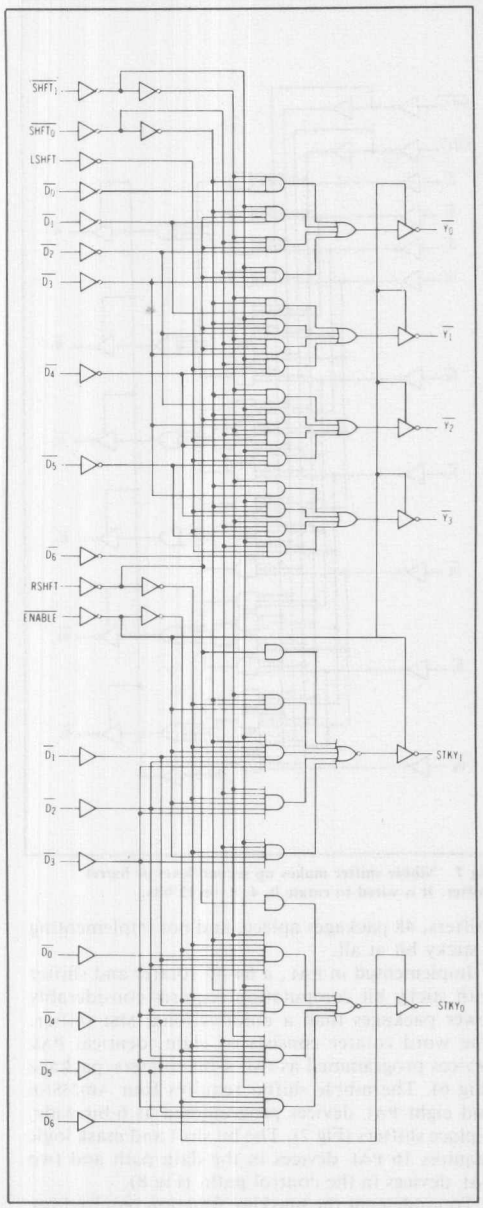


Fig 8 Bit shift and mask logic for barrel shifter requires 16 PAL devices in the data path and two PAL devices in the control path. Each shift and mask PAL device has an enable input from one mask control PAL device and is connected to enable inputs of its left and right neighbors. The 64-bit masking operation requires only 16 control lines.

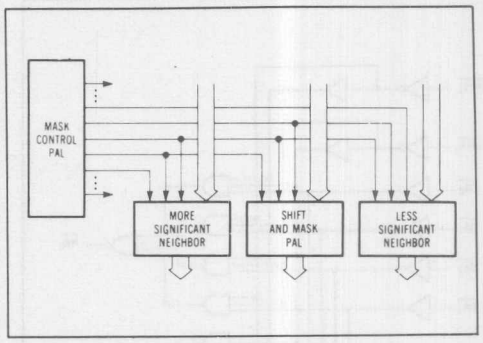


Fig 9 "Nearest Neighbor" interconnection is used to implement masking function for shifting. A mask control PAL device determines if all 4 bits are to be masked; enables from the adjacent slice determine if the PAL device is at shift boundary.

connected to the enable inputs from its left and right-hand neighbors (Fig 9). The mask control PAL input determines if all 4 bits from the slice should be masked off. Enables from the adjacent slice determine if a PAL unit is at a shift boundary. If only one of the neighboring slices is disabled, the shift and mask PAL unit masks off from 0 to 3 of the bits adjacent to the disabled slice, depending on the bit-rotation distance. In this way, the 64-bit masking operation can be implemented with only 16 control lines as opposed to at least 64 for an SSI/MSI solution.

In addition to performing the final shift and mask operation, the bit-shifter PAL unit also computes the logical OR of the masked-out bits at each slice position. These outputs are then logically ORed to generate a sticky bit. The extra hardware required is less than two SSI packages. The entire PAL barrel shifter requires 38 devices to implement 64-bit rotation, left shifting, right shifting, and sticky bit accumulation. An MSI-based left/right shifter, without sticky bit computation, requires a minimum of 96 parts. In addition, the logic required for implementing the prescale and normalize operations is significantly reduced through the use of PAL devices.

New PAL® Device Architecture Extends Design Flexibility

CP-155

Introduction

The introduction of Programmable Array Logic (PAL®) devices by Monolithic Memories in the late 1970s marked the beginning of rapid growth for the programmable logic device market. By offering fast turn development of user-defined circuit functions and significant package count reduction, programmable logic devices (PLDs) solved such system design problems as time to market, system size and cost, and the cost of making design changes.

First generation PLDs, such as the PAL16L8, PAL16R4, PAL16R6, and PAL16R8, offer relatively simple, single-function architectures. Designers choose either devices with combinatorial outputs, registered outputs, or a combination of both, to solve their specific design tasks.

While simple architectures still offer the fastest operating speeds, more universal architectures have emerged among second generation PAL devices, which allow more user definition of the device configuration. The universal "macrocell" approach allows users to reduce the number of different parts in a system and enjoy considerable flexibility in defining the final PLD architecture.

All of these devices enable logic designers to combine the tasks of several SSI devices into a single PLD. System functions in the form of small state machines or simple data flow machines are possible in a single PLD. But the ability of designers to build significant "systems-on-a-chip" has been somewhat hampered either by a lack of device flexibility or the inability to make full use of the logic power of the PLD. This has kept the benefits of PLDs inaccessible to some designers. The purpose of this paper is to describe some of the key features of a new PAL device which opens up a whole range of possibilities by removing many of these limitations.

The PAL32VX10

The new PAL32VX10 takes the synchronous logic approach used by the PAL22V10. Judicious improvements to the macrocell provide logic power far greater than anything currently available in 24 pins. This is accomplished while maintaining a 25-ns propagation delay (t_{PD}) with only 180-mA current drain.

The PAL32VX10 shares some of the features of the PAL22V10, such as varied product term distribution, and can emulate any PAL22V10 function. It is therefore appropriate to describe the individual features of the PAL32VX10 by contrasting them with the corresponding more limited options in the PAL22V10. The key features to be discussed are:

1) Dual Feedback
2) Programmable Flip-Flops
3) Register Bypass Scheme

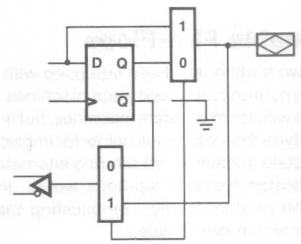
After discussing these features we can look at the overall power provided by the integration of these features into a single PLD.

Adapted from a paper presented at Northcon/86

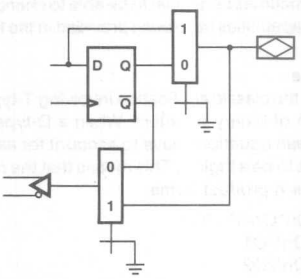
Dual Feedback

The feedback scheme used in the PAL22V10 (Figure 1a) provides two sources for the feedback signal, only one of which is used at any given time. When the flip-flop is used, the feedback signal is taken from the output of the flip-flop itself; when the flip-flop is bypassed, the feedback signal comes from the output pin, allowing for bidirectional signal flow.

There are two basic limitations in this approach. The first is the fact that registered outputs cannot be made bidirectional. One could add a separate feature allowing such an option (Figure 1b), but such an improvement by itself provides an incremental benefit. The second limitation, which is substantial from the standpoint of device utilization, is the fact that if an output is needed as an input, the output macrocell for that pin is completely wasted.



(a) As It Currently Exists



(b) A Possible, But Marginal Improvement

Figure 1. PAL22V10 Feedback Selection Scheme

The approach taken in the PAL32VX10 is to provide two completely independent feedback lines: one from the flip-flop, and one from the output pin (Figure 2). That this solves the first limitation is obvious; the output can always be disabled and used as an input without interfering with the signal being fed back from the flip-flop.

PAL® is a registered trademark of Monolithic Memories.

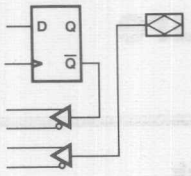


Figure 2. PAL32VX10 Dual Feedback Scheme

But the benefits go far beyond that. Using the output pin permanently as an input does not require that the macrocell be wasted: the flip-flop can be "buried", allowing internal state machines to be built. This removes the second limitation of the PAL22V10 scheme. Alternatively, the macrocell can be used as an input register by feeding the input into the flip-flop, and then using the flip-flop feedback to drive other logic in the device. This same approach can also be used when the pin is used as an output, allowing the flip-flop to be loaded directly from its output pin.

Thus the thirty-two inputs to the array consist of twelve dedicated inputs, ten output pin feedback inputs, and ten flip-flop feedback inputs.

Programmable Flip-Flops

PAL devices have traditionally been equipped with D-type flip-flops for data synchronization and state machines. D-type flip-flops work quite well for many state machines, but in many other cases, J-K or T-type flip-flops would allow for implementation of more complex state machines. Yet offering alternative products with other dedicated types of flip-flops would unnecessarily overpopulate the product family, complicating the designer's device selection and inventory tasks.

The ideal solution is to have a single device which can have each flip-flop individually programmed to define its type. And, as will be seen, it is sometimes beneficial to be able to change types "on the fly". These capabilities have been provided in the PAL32VX10.

T-type flip-flops

As an example, the classic application involving T-type flip-flops is in the design of binary counters. When a D-type flip-flop is used, the Boolean equations have to account for each time the device output is to be a logic 1. This means that the nth bit of the counter requires n product terms:

```
Qn+ := /Qn*Qn-1*...*Q1
     + Qn*/Q1
     + Qn*/Q2
        .
        .
        .
     + Qn*/Qn-1
```

By using a T-type flip-flop, one only has to determine when the flip-flop output should toggle. The nth bit toggles only when all lower order bits are "1". This can be expressed as

"Qn should HOLD UNLESS all lower order bits are HI."

The UNLESS operator can be implemented by an Exclusive-OR (XOR) gate:

```
Qn+ := Qn :+: (Qn-1*Qn-2*...*Q1)
```

This requires two product terms (one on each input of the the XOR gate), regardless of which bit of the counter is being defined.

So by providing an XOR gate with Q fed back to one input, the combination XOR gate/D-type flip-flop acts as a T-type flip-flop. A sum of products on the other XOR input provides the Toggle signal.

J-K flip-flops

The classic formula for a J-K flip-flop is

```
Q+ := J*/Q + /K*Q.
```

This expression can be directly implemented in a D-type flip-flop, providing /K fits within the available product terms. If this is not the case, or if finding /K is inconvenient, another formulation is needed. We know that by the definition of a J-K flip-flop,

Q should HOLD UNLESS:
the output is 0 and we SET to 1;
the output is 1 and we RESET to 0;
we are toggling (JK = 11).

The above conditions for "not holding" can be plugged into a Karnaugh map and minimized to give the following expression:

```
Q+ := Q :+: (J*/Q + K*Q).
```

Here, with the help of the XOR gate, a J-K flip-flop is implemented directly in terms of J and K, instead of /K. Again, this is done by feeding Q to one input of the XOR gate, and by ORing several product terms into the other XOR input. These terms are divided between J terms and K terms; J terms are ANDed with /Q, while K terms are ANDed with Q.

So we can implement J-K, T, and D-type flip-flops by using the XOR gate in combination with a D-type flip-flop. By setting one XOR input to Q, /Q, 1, or 0, we can implement J-K, /J-/K, J-/K, or /J-K flip-flops. The other XOR input provides a sum of products for J and K. Of course, since S-R flip-flops are a subset of J-K flip-flops, they can be implemented in the same fashion.

Reconfigurable flip-flops

This is a step in the right direction, but one can do more. Consider the design of a loadable counter. This is a situation where a T-type flip-flop is preferred when counting, but a D-type is better when loading. A T-type configuration can be converted to D-type, however, if the LOAD control input is used to disable the XOR input containing Q.

Thus by making the XOR input a product term, instead of a dedicated single input, the flip-flop type can be changed on the fly to allow each particular function to be implemented as efficiently as possible. Note that PLDs which provide programmable flip-flops without the XOR control product term cannot have their flip-flops altered once configured. So, for instance, adding a LOAD feature to a counter then requires twice as many extra product terms as required with a reconfigurable flip-flop.

All of these considerations have been accommodated in the PAL32VX10 (see Figure 3). The single XOR product term provides control of the flip-flop type, for either permanent or reconfigurable flip-flops. From eight to sixteen product terms are available as D or T inputs, or to be divided between J and K inputs. Some of the possible configurations are shown in Figure 4.

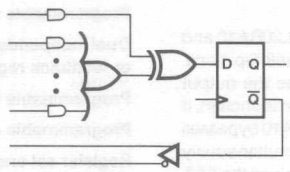
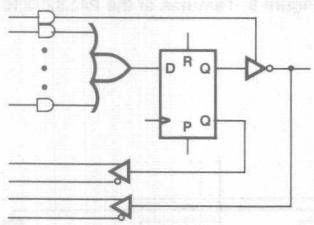
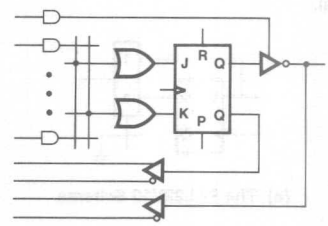


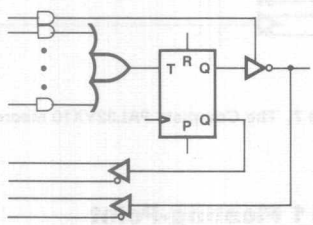
Figure 3. The PAL32VX10 Programmable Flip-Flop



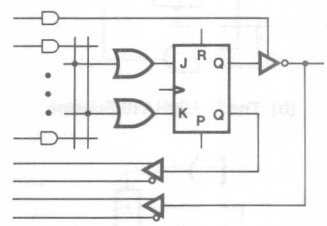
D Register with I/O



J-K Register with I/O



T Register with I/O



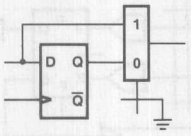
Buried Register with Dedicated Input

Figure 4. A Few Selected Flip-Flop Configurations

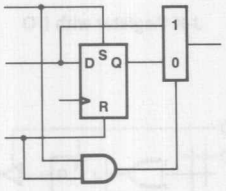
Register Bypass

The two primary second-generation PLDs, the PAL20RA10 and the PAL22V10, have different ways of bypassing their flip-flops. The PAL22V10 has a single fuse which configures the output permanently (see Figure 5a). Once the configuration is chosen, it cannot be changed. On the other hand, the PAL20RA10 bypasses its flip-flop by activating both SET and RESET simultaneously (see Figure 5b). This can be done dynamically, but since the SET and RESET affect the contents of the flip-flop, one cannot, for instance, clock data into the flip-flop, change to combinatorial, and then change back to look at the flip-flop; the data in the flip-flop will have been lost.

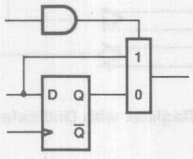
The PAL32VX10 eliminates this restriction by using a product term to control the bypass multiplexer (see Figure 5c). While the output is combinatorial, the flip-flop is unaffected. This makes it possible to use the flip-flop in macrocells that are sometimes combinatorial.



(a) The PAL22V10 Scheme



(b) The PAL20RA10 Scheme



(c) The PAL32VX10 Scheme

Figure 5. Register Bypass Schemes

The Complete Macrocell

This has not been an exhaustive description of all of the features of the PAL32VX10 macrocell, a summary of which is shown in Figure 6. This paper has merely highlighted the more revolutionary capabilities. The complete macrocell is shown in Figure 7. All of the elements work together to offer unparalleled flexibility, without compromising the performance of the device.

An application will serve to illustrate the power afforded by all of these features operating in concert. It is beyond the scope of this paper to describe the details; a brief description will suffice here.

Programmable registered or combinatorial outputs

Dual independent feedback paths allow I/O with combinatorial or feedback register outputs

Programmable flip-flops allow J-K, S-R, T or D types

Programmable output polarity

Register set and register reset can be asynchronous or synchronous

Automatic register preset on power up

Varied product term distribution

Up to 16 product terms per output

Figure 6. Features of the PAL32VX10

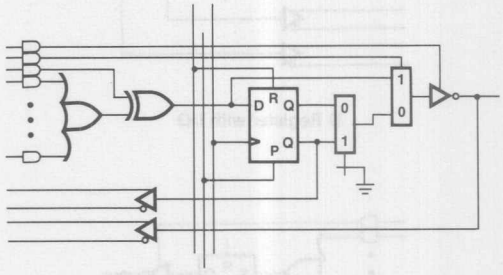


Figure 7. The Complete PAL32VX10 Macrocell

A PDP-11 Floating-Point Coprocessor Controller

This device monitors a PDP-11 bus for instructions that should be sent to a coprocessor. As shown in Figure 9, it provides the PDP instruction decode, an instruction tracking state machine which provides various wait states for operand fetching, and coprocessor instruction encoding. The 3-bit state machine is not needed externally; thus it is buried, and the output pins provide additional inputs. Reset capability and a test feature for examining the buried flip-flops are included.

This application takes advantage of the varied product term distribution, the buried flip-flops, and J-K, T, and D-type flip-flops. To implement the same application without a PAL32VX10 would require:

1 PAL22V10
1 PAL16R4
1 PAL14H4

Of course, since we are talking about replacing several PAL devices with another PAL device, the number of discrete TTL chips that are being replaced is very large indeed.

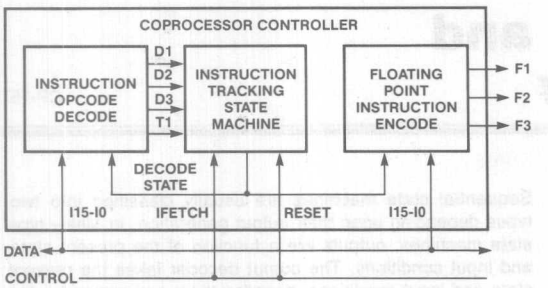
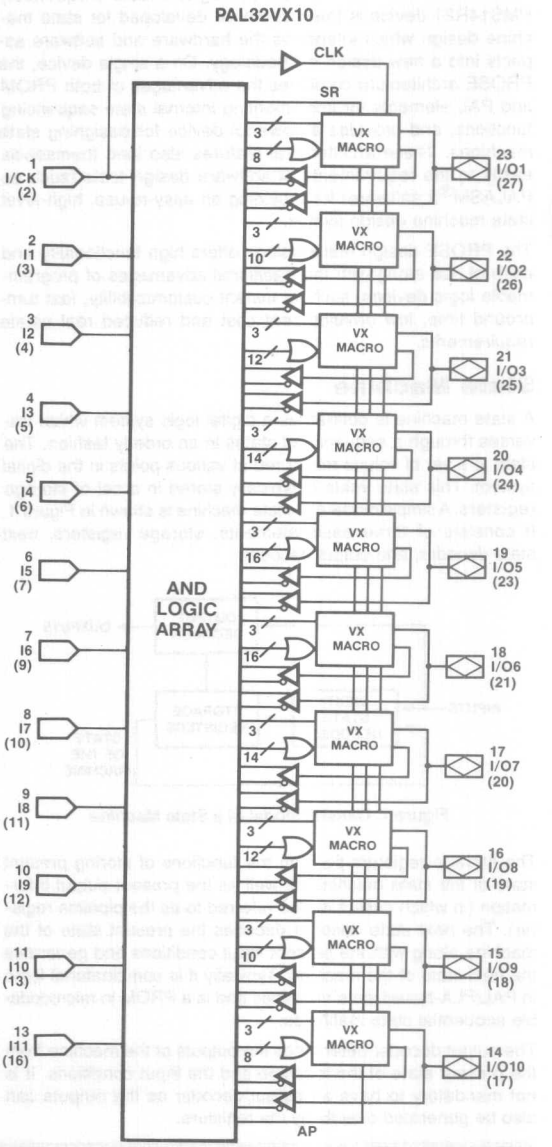


Figure 8. Coprocessor Controller

Summary

The PAL32VX10 provides significantly increased flexibility as a result of a few carefully placed improvements. The addition of another feedback line, an XOR gate, and a few other enhancements allows full utilization of the device. The utilization can be further optimized through the ability to represent the desired functions in the most compact form possible. These capabilities combine to form the most versatile 24-pin PLD available to date.



PROSE Architecture and Design Methodology

CP-157

Introduction

With increasing density and functionality, programmable logic devices are being used for such specialized applications as state machines. The PROSE (PROgrammable SEquencer) PMS14R21 device is the first device developed for state machine design which integrates the hardware and software aspects into a new design methodology. On a single device, the PROSE architecture combines the advantages of both PROM and PAL elements for implementing internal state sequencing functions, and provides a powerful device for designing state machines. These architectural features also lend themselves easily to the requirements of software design tools (such as PALASM®2 software) for providing an easy-to-use, high-level state machine design syntax.

The PROSE design methodology offers high functionality and ease of use along with the traditional advantages of programmable logic devices, such as instant customizability, fast turnaround time, low development cost and reduced real estate requirements.

State Machine

A state machine is defined as a digital logic system which traverses through a sequence of states in an orderly fashion. The state is a set of values measured at various points in the digital system. This state value is typically stored in a set of storage registers. A simple model of a state machine is shown in Figure 1. It consists of three basic elements: storage registers, next state decoder, and output decoder.

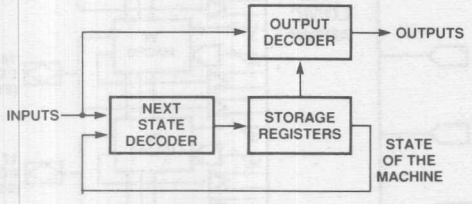


Figure 1. General Model of a State Machine

The storage registers perform the functions of storing present state of the state machine as well as the present output information (in which case it is also referred to as the pipeline register). The next state decoder decodes the present state of the machine along with the present input conditions and generates the next state of the machine. Typically it is combinatorial logic in PAL/PLA-based state machines and is a PROM in microcodable sequential state machines.

The output decoder determines the outputs of the machine from the present state of the machine and the input conditions. It is not mandatory to have an output decoder as the outputs can also be generated directly by the registers.

Sequential state machines are usually classified into two types depending upon their output generation. In Mealy type state machines, outputs are a function of the present state and input conditions. The output decoder takes the present state and input conditions, decodes them and generates the outputs. In Moore type state machines, outputs are strictly a function of the state of the machine. Thus, the Moore machine usually requires a simpler output decoder to decode the present state only, or none at all in a case where the storage registers can be used to provide these outputs directly.

As transitions to the next state are dependent upon the present state and input conditions, the operation of a Mealy machine can be emulated by a Moore machine, provided that one cycle delay is allowed between input conditions and output generation.

State Machine Representation

The first step in the design methodology requires converting the design problem into its state diagram representation. The state diagram shown in Figure 2 consists of state identifiers, input conditions and the outputs generated. The states are represented by bubbles, with arrows indicating transitions to other states. The input conditions as well as outputs generated are associated with these transitions.

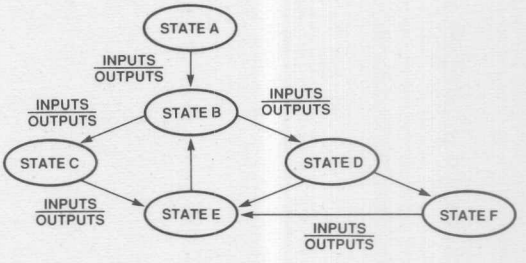


Figure 2. A Typical State Diagram

There are two major operational requirements for state machines as shown in Figure 3.

- Control sequencing
- Output generation

Control sequencing is changing from one state to another in a state machine. If the transition from the present state to the next state is dependent upon the present state, it is direct control sequencing. If the transition from the present state to the next state is dependent upon the present state and the input conditions, it is called conditional control sequencing.

Adapted from a paper presented at Electro/87.

As mentioned earlier, the output generation can be based upon the present state for a Moore type of state machine, or the present state and inputs for a Mealy type of state machine. The state diagram representation includes both the control sequencing and output generation representations.

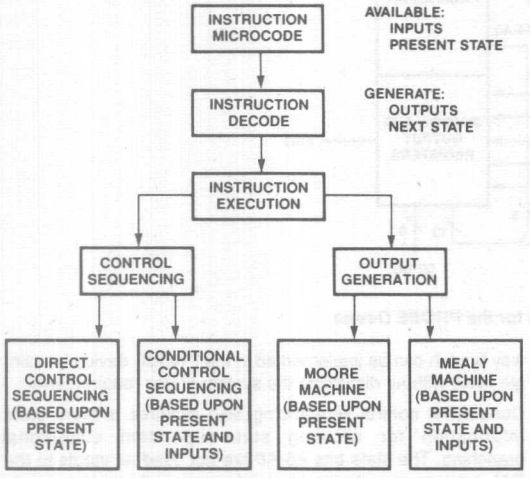


Figure 3. Instruction Execution in State Machines

The design process involves converting the hardware problem to this state machine representation. The system design is analyzed for its signal logic and timing requirements. The outputs generated are based upon the system signal logic requirements, and the state transitions depend upon the system timing requirements of these signals.

Traditional Microprogrammable Sequencers

Traditional high-end microprogrammable sequencers fit this general model of state machines. They use simple clocked D-type registers as pipeline registers as shown in Figure 4. Typically they do not have an output decoder and allow only Moore machine implementations.

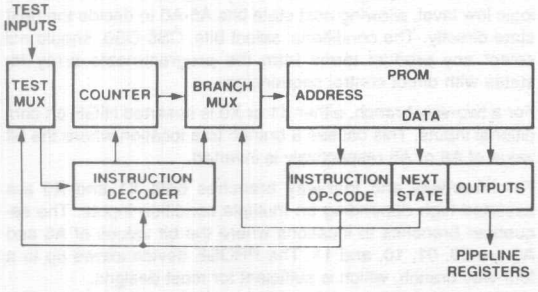


Figure 4. Traditional Microprogrammable Sequencer

For the next state decoder, microprogrammable sequencers use a PROM array. Each location of the PROM functions as a state. PROMs have been traditionally used for this purpose because of their familiarity and relative ease of use. This PROM is addressed by the pipeline register field storing the present state. The PROM data bits are inputs to the pipeline register which are loaded at every clock cycle. The PROM data consists of the next state (branch), output logic values, and an instruction op-code.

Output generation in microprogrammable sequencers is the direct transfer of PROM data to the output field of the pipeline register on every clock cycle.

For direct control sequencing, next state values from the PROM are loaded into the pipeline registers at every clock cycle, representing a state transition. In most microprogrammable sequencers, a binary counting mechanism is also added to the state registers. This allows the PROM to be addressed with a state counter. This is effectively control sequencing with serial binary state numbers. The PROM next state value is used only when a non-serial branch is required. A branch select multiplexer is used to select between this serial count and the branch for the next state of the machine.

For conditional control sequencing an input condition testing multiplexer is used. Usually it tests for one of the inputs at logic high or low. Based on this input condition, the branch select multiplexer is controlled, providing conditional branching.

The instruction op-code stored in the PROM for every state controls the sequencer operation for that state. It decides between direct and conditional control sequencing. It also decides the input condition to be tested. These instruction op-codes are stored in the data form and are decoded just before execution. Extra hardware is required to decode these instruction op-codes and generate internal multiplexer control signals. This slows down the device operation by the time required to decode the instructions.

The combination of instruction op-codes, output, and next state values is called the instruction microcode. The design process requires writing instruction microcode based upon the state diagram. This approach to a microprogrammable sequencer, though popular, imposes several restrictions on the state machine operation as the instruction op-codes are predefined and fixed. These fixed op-codes can only perform certain operations and the state machine design is usually modified to fit those restrictions. This means that the sequencer operation is usually not optimal for the system design problem.

For example, condition testing, which is restricted to one input testing only, may not fit a design requiring simultaneous testing of two signals. The branching is also restricted to a two-way branch. For multi-way branching, a microprogrammable sequencer requires multiple cycles which could disrupt the system signal timing requirements. In some devices further dedicated hardware is added to improve the functionality of the sequencer. Such hardware is usually under the control of instruction op-code and provides features such as testing more than one input or branching to more than one next state.

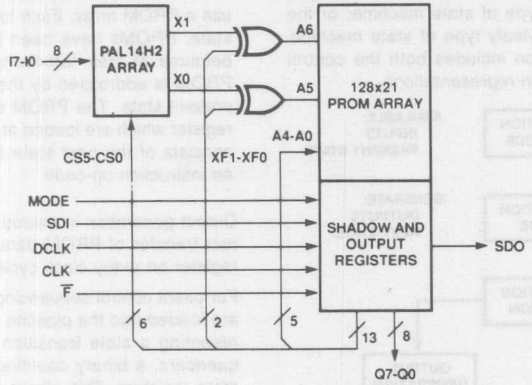


Figure 5. A Block Diagram for the PROSE Device

PROSE Device Architecture

The PROSE device architecture is very similar to this model of a state machine. Its architecture (Figure 5) is optimized to perform the functions of a sequential state machine as represented in the model. For the storage registers, the PROSE device provides a set of twenty-one pipeline registers. These registers are used to store the present state and the outputs generated by the machine. This is similar to the microprogrammable sequencers. Beyond this however, the similarity ends. The architecture of the PROSE device has been defined specially to optimize its functionality.

The next state decoder in the PROSE device consists of a PROM. This PROM stores the next state, outputs and branch control information. The output generation is similar to microprogrammable sequencers, and is a direct transfer of PROM data to the output field of the pipeline register. There are eight outputs in the PROSE PMS14R21 device.

Control Sequencing

Direct control sequencing in the PROSE device is implemented by the PROM. The depth of the PROM is 128, allowing up to 128 states. Seven bits (A0-A6) of the pipeline register's next state field are used to address the PROM to provide the state transition capability. For direct control sequencing the next state values from the PROM are loaded into the pipeline register's next state field, for a state transition.

Conditional Control Sequencing

For conditional control sequencing, a combinatorial PAL array is provided. All of the eight conditional inputs I0-I7 are direct inputs to this PAL array. This PAL array provides two major advantages over microprogrammable sequencers:

- It allows selection of a combination of input signals for condition testing. This is very useful for many system designs which require testing of a combination of two or more signals at particular logic levels. This capability is not available in most microprogrammable sequencers.
- It allows multiple condition testing for multiple branching. Microprogrammable sequencers require more than one cycle to implement multiple branching on multiple conditions. A four-way branch can be implemented in the PROSE device in a single cycle without disrupting the system timing requirements.

Conditional control sequencing also requires present state information for allowing state-dependent conditional branching. The state bits A0-A6 are not used as inputs to the PAL array to provide the state information. An easier way to provide state information to this PAL array is to use extra bits from the pipeline registers and PROM as its inputs. These six bits, CS0-CS5, form the condition select field, which allows user-definable bit combinations to be used for any state. This optimizes the use of the PAL array product terms, since the same bit combinations can be used for testing the same input conditions for different states.

Branching

To implement branching, the two outputs of this PAL array, X0 and X1, are XORed with the two most significant bits of the state address (XF1 and XF0) from the pipeline register. This gives a capability of inverting state bits A5 and A6 depending upon the condition testing in the PAL array. This conditional change of next state address provides a four-way branch capability.

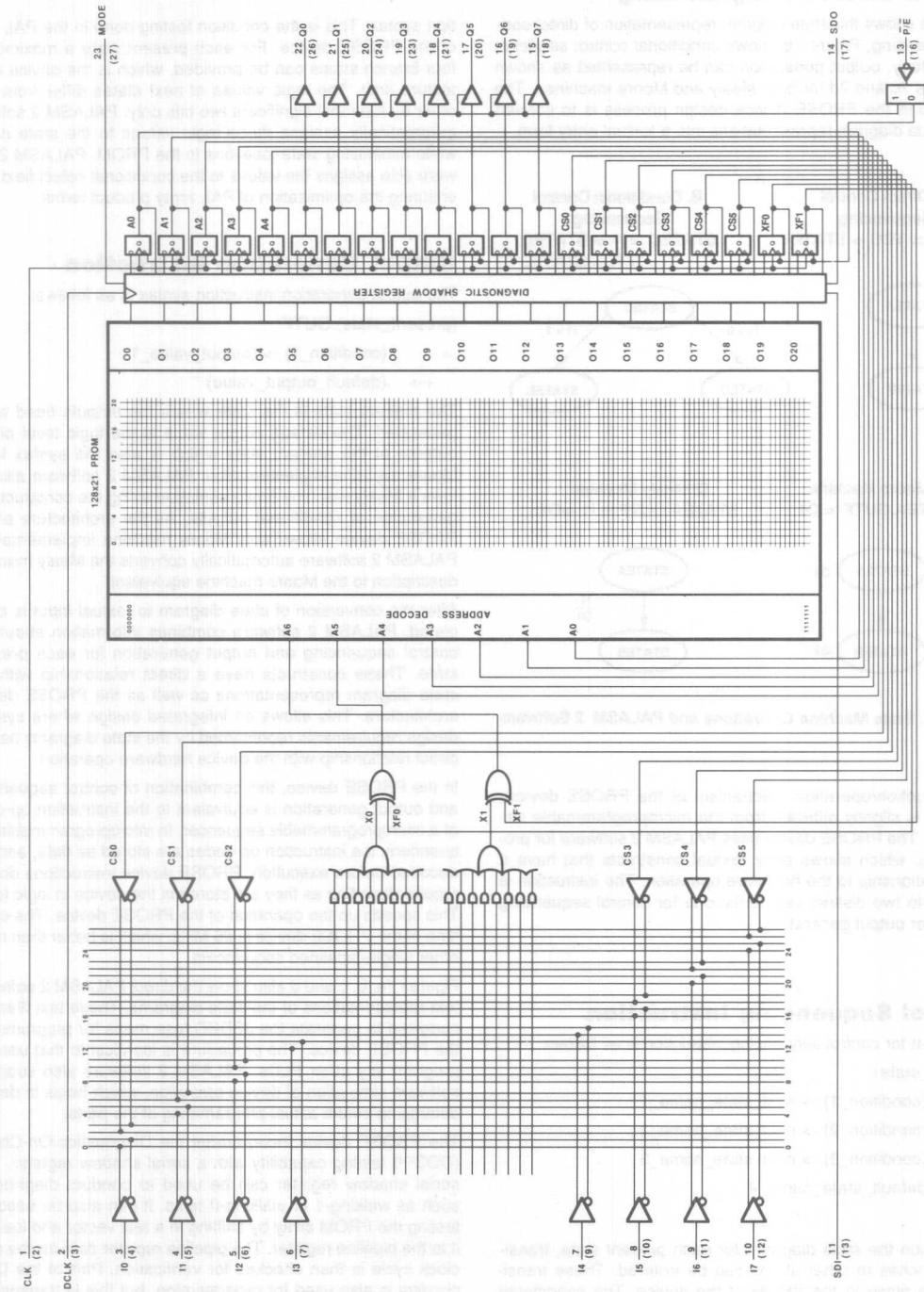
For direct control sequencing, no product term in the PAL array is selected. The programmable array outputs X1 and X0 are at logic low level, allowing next state bits A6-A0 to decide the next state directly. The conditional select bits, CS5-CS0, should not select any product terms from the programmable array for states with direct control sequencing.

For a two-way branch, either X1 or X0 is asserted HIGH on conditional inputs. This causes a branch to a location where the bit value of A6 or A5 respectively is inverted.

For three-way and four-way branches both X1 and X0 are asserted high depending on multiple condition inputs. The sequencer branches to locations where the bit values of A6 and A5 are 00, 01, 10, and 11. The PROSE device allows up to a four-way branch, which is sufficient for most designs.

In the PROSE device, the next state decoder has been effectively partitioned into a PROM and a PAL device, to utilize the efficiencies of the two for control sequencing and conditional testing respectively.

Logic Diagram



PROSE Device Programming

Figure 7a shows the state diagram representation of direct control sequencing; Figure 7b shows conditional control sequencing. Similarly, output generation can be represented as shown in Figures 7c and 7d for both Mealy and Moore machines. The next step in the PROSE device design process is to convert these state diagram representations into a textual entry form.

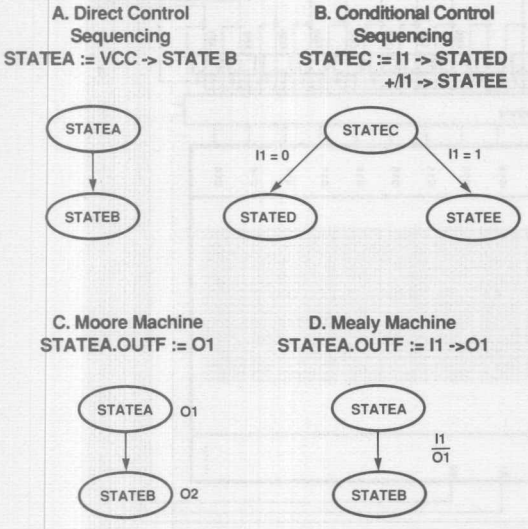


Figure 7. State Machine Operations and PALASM 2 Software

The instruction/operation mechanism of the PROSE device, however, is slightly different from the microprogrammable sequencers. The PROSE device uses PALASM 2 software for programming, which allows easy textual constructs that have a direct relationship to the hardware operation. The instruction is divided into two distinct segments: one for control sequencing and one for output generation.

Control Sequencing Instruction

The format for control sequencing instruction is as follows:

```
(present_state)
:=      (condition_1) -> next_state_name_1
  +     (condition_2) -> next_state_name_2
  +     (condition_3) -> next_state_name_3
  +->   default_state_name_4
```

Based upon the state diagram, for each present state, transitions/branches to other states can be entered. These transitions take place in the PROM of the device. The conditional inputs can also be described in the condition field of the instruction syntax. This is the condition testing done in the PAL array of the PROSE device. For each present state a maximum of four branch states can be provided, which is the device architecture limit. The logic values of next states differ from each other by the most significant two bits only. PALASM 2 software automatically assigns these logic values to the state names while minimizing state locations in the PROM. PALASM 2 software also assigns the values to the conditional select field while ensuring the optimization of PAL array product terms.

Output Generation Instruction

The output generation instruction syntax is as follows:

```
(present_state).OUTF
:=      (condition_1) -> (output_value_1)
  +->   (default_output_value)
```

The present state is the state where the outputs need to be generated. The default output value is the logic level of the outputs at the present state which is also the syntax for a Moore machine implementation. PALASM 2 software also allows a Mealy machine implementation using the constructs for generation of conditional outputs. As the architecture of the PROSE device allows only Moore machine implementation, PALASM 2 software automatically converts the Mealy machine description to the Moore machine equivalent.

After the conversion of state diagram to textual input is completed, PALASM 2 software combines information about the control sequencing and output generation for each present state. These constructs have a direct relationship with the state diagram representations as well as the PROSE device architecture. This allows an integrated design where system design requirements represented by the state diagrams have a direct relationship with the device hardware operation.

In the PROSE device, this combination of control sequencing and output generation is equivalent to the instruction op-code of a microprogrammable sequencer. In microprogrammable sequencers, the instruction op-codes are stored as data, and are decoded before execution. PROSE device instructions do not require decoding as they are stored in the device in logic form. This speeds up the operation of the PROSE device. The cycle time of the PROSE device is 25 MHz, which is better than most other single-pipelined sequencers.

Figures 7a, b, c and d also show the direct PALASM 2 software text representations of the state diagrams. These text files are compiled to generate the JEDEC fuse maps for programming the PROSE device. The procedure is identical to that used to program any other PLDs. PALASM 2 software also supports software simulation of device operation, which helps in design debugging before actual programming of the parts.

The PROSE device incorporates the Diagnostics-On-Chip™ (DOC™) testing capability with a serial shadow register. This serial shadow register can be used to conduct diagnostics such as walking-1 or walking-0 tests. It can also be used for testing the PROM array by shifting in a test vector and loading it to the pipeline register. The pipeline register data for the next clock cycle is then checked for verification. Part of the DOC circuitry is also used for programming, but this is transparent to the users.

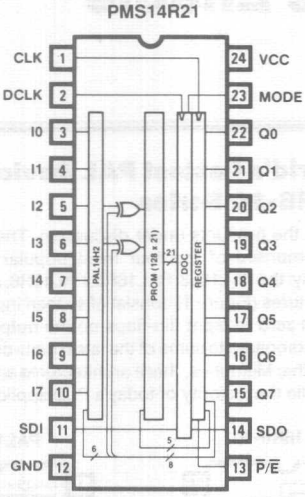


Figure 8. DIP Pin Configuration

Summary

With continuing advances in technology, new devices are being introduced which are tailored to specific system design requirements. The PROSE device is one such device, with an architecture optimized for state machine design. It combines a PROM array and a PAL array on a single device, and utilizes the functional advantages of both for control sequencing and conditional testing respectively. This combination offers higher functionality over the traditional approach of microprogrammable sequencers for state machines. It also offers a newer design methodology where software instructions are split into control sequencing and output generation instructions. These instructions are directly related to hardware requirements, unlike the instruction op-codes of traditional microprogrammable sequencers, which sometimes require design compromises. The PROSE device also offers extensive simulation, thus offering a complete solution.

2

Blazing Fast PAL® Devices Enable New Application Areas

CP-156

The PAL16R8-10 Series from Monolithic Memories is the fastest TTL programmable logic family, and the ECL PAL devices from Monolithic Memories are the fastest programmable logic devices of any technology.

Do these devices simply drop into the same applications as previous versions? No! These high-speed PAL devices offer several advantages, including speed, that open up whole new applications for programmable logic devices.

The World's Fastest PAL Devices PAL16R8-10 Series

Let's look at the products under discussion. The PAL16R8-10 Series is comprised of the four most popular PLDs in the industry today; the PAL16L8, PAL16R8, PAL16R6, and PAL16R4. The architectures (Figure 1) consist of sixteen inputs and eight outputs, with zero to eight flip-flops on the outputs. Although very simple compared to some of the latest high-density devices from Monolithic Memories, these architectures are very powerful, and handle the majority of today's PLD applications.

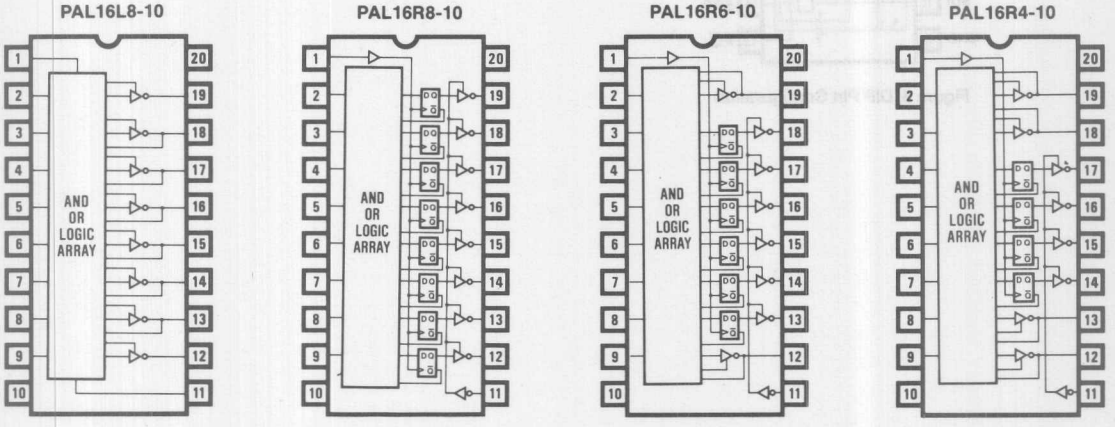
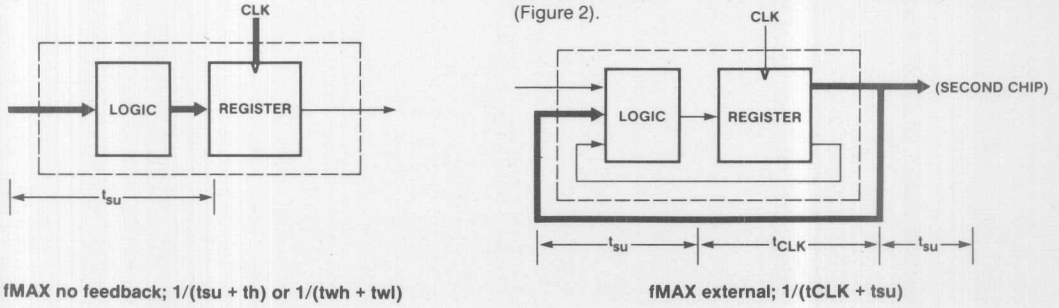


Figure 1. The Four Devices Comprising the PAL16R8-10 Series. Each Has Eight Product Terms per Output.

The speed of the PAL16R8-10 Series is 10 nanoseconds (ns) maximum from input to output on any combinatorial path. The registered paths offer a 10-ns setup time and only 8 ns from clock input to data output. This translates into a maximum clock frequency (fMAX) of 1/(10 ns + 8 ns), or 55.5 MegaHertz (MHz), if external feedback is used from the flip-flop.

It is important to differentiate this number from another common fMAX calculation, the maximum flip-flop toggle rate. The toggle rate is limited by the greater of the data setup time or the clock widths high and low. The toggle rate specification for the PAL16R8-10 Series is 62.5 MHz. However, this specification is unrealistic since it does provide for feedback. It is a more representative path that provides the 55.5-MHz specification (Figure 2).



fMAX no feedback; 1/(tsu + th) or 1/(twh + twl)

fMAX external; 1/(tCLK + tsu)

Figure 2. Alternative Methods of Measuring fMAX. A More Realistic Path Is the Second One, Including External Feedback.

Adapted from a paper presented at Electro/87.

ECL PAL Devices

The PAL10H20P8 from Monolithic Memories was the first ECL PAL device. This device is combinatorial, with twenty inputs and eight outputs. A sequential device followed, the PAL10H20G8, with the same number of inputs and outputs, but having the added feature of output latches (Figure 3).

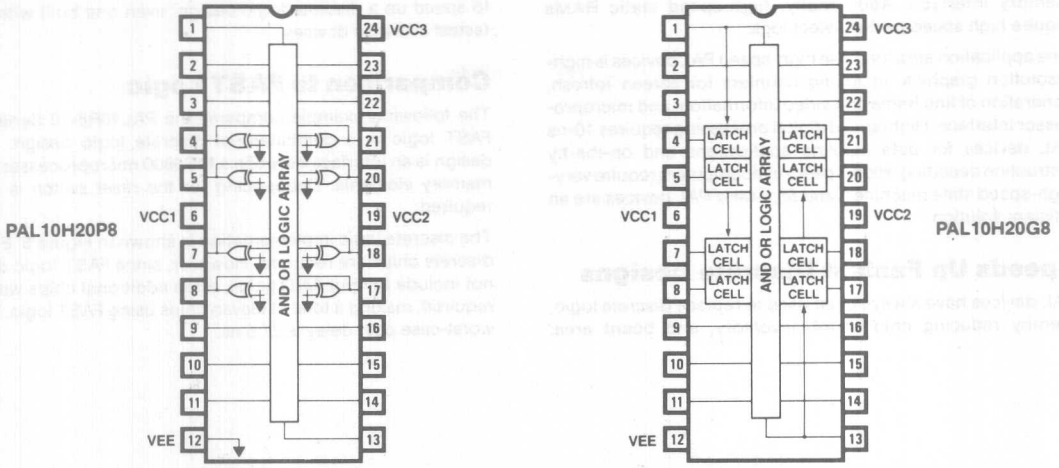


Figure 3. The ECL PAL Devices. On the Left Is the PAL10H20P8 Combinatorial Device, and on the Right Is the PAL10H20G8 Latched Device.

The PAL10H20P8 and PAL10H20G8 have combinatorial propagation delays of only 6 ns, faster than any other PLDs. The latched version has a setup time of 4.5 ns and a latch-to-output delay of only 2.5 ns. This provides a maximum frequency of over 142 MHz.

History of Speed Developments

A short history of speed improvements in bipolar PAL devices shows how these devices compare to earlier versions (Figure 4). In this graph of speed vs. power, performance characteristics closer to the lower left are desired. The first PAL devices, introduced by Monolithic Memories in the late 70s, were in the 35-ns range, and consumed 180 mA. Technology improvements allowed the introduction of the first 25-ns devices in 1983. In early 1985 the first 15-ns devices were released, with half-power versions at 25 ns and quarter-power versions at 35 ns. Last year Monolithic Memories introduced its industry-leading 10-ns PAL16R8-10 Series, along with the ECL devices.

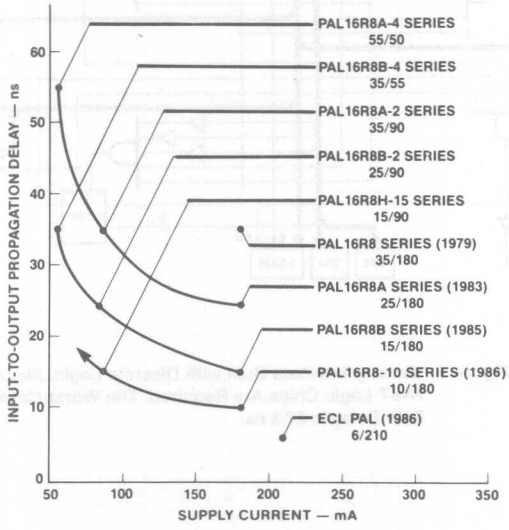


Figure 4. Speed vs. Power for Monolithic Memories Bipolar PAL Families. The Year of Introduction Is Noted. The PAL16R8-10 Series and the ECL PAL Devices Offer the Highest Performance in the Industry.

New High-Performance Applications

The raw speed of these new devices allows their use in new high-performance applications. As microprocessors reach 20 MHz and beyond, 50 MHz logic is required for peripheral and memory interface. Additionally, high-speed static RAMs require high speeds in the select logic.

One application area for these high-speed PAL devices is high-resolution graphics, including counters for screen refresh, generation of line frame and video information, and microprocessor interface. High-speed digital processing requires 10-ns PAL devices for data pipeline applications and on-the-fly instruction decoding. High-speed minicomputers require very-high-speed state machines, and 55.5-MHz PAL devices are an efficient solution.

Speeds Up Fastest Discrete Designs

PAL devices have always been used to replace discrete logic, thereby reducing chip count, inventory, and board area; simplifying board design and layout; and adding extensive design flexibility. Now PAL device speed has broken an important barrier; the PAL16R8-10 Series allows PAL devices to speed up a discrete logic design, even one built with the fastest available devices.

Comparison to FAST Logic

The following example compares the PAL16R8-10 Series to FAST logic in a combinatorial discrete logic design. The design is an interface between a MC6800 microprocessor and memory elements. Debouncing for the reset switch is also required.

The discrete logic implementation is shown in Figure 5. Eight discrete chips are required. However, since FAST logic does not include 8-input AND gates, three additional chips will be required, making a total of eleven chips using FAST logic. The worst-case path delay is 27.6 ns.

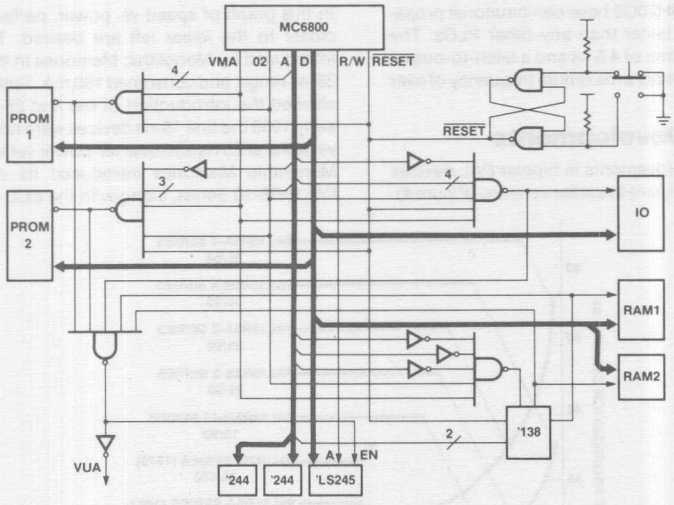


Figure 5. Memory Interface Built with Discrete Logic. Eleven FAST Logic Chips Are Required. The Worst-Case Path Delay Is 27.6 ns.

Figure 6 shows the same design implemented in a PAL16L8-10. One device replaces eleven chips, and the new worst-case path delay is now only 20 ns.

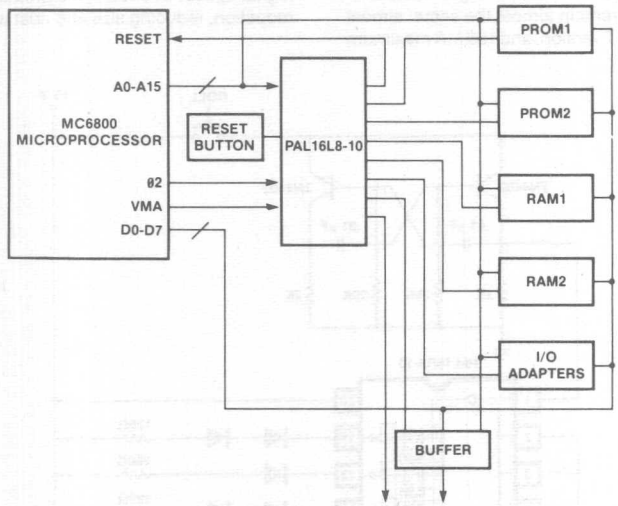


Figure 6. Same Memory Interface Implemented in a PAL Device. Only One Chip Is Required, and the Delay Is Only 20 ns Maximum.

An example of a sequential circuit (Figure 7) is a simple dice game, providing two die readouts on LEDs. The discrete logic design requires seven chips. The worst-case path for FAST logic has a maximum frequency of 27.5 MHz.

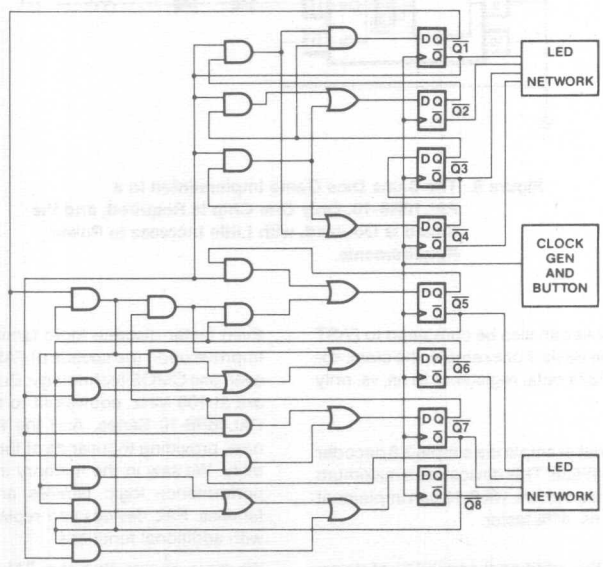


Figure 7. Simple Dice Game Constructed from Discrete Gates. Seven Chips Are Required, with a Maximum Frequency of 27.5 MHz.

The entire design can be implemented in one PAL16R8-10 (Figure 8). The worst-case path has a maximum frequency of 55.5 MHz, over twice as fast as the discrete logic version. In addition, power requirements remain almost the same: almost 170 milliAmps (mA) for the FAST version, and 180 mA maximum for the PAL device. A half-power version of the PAL device will require 47% less power than FAST logic while still providing higher speed. Of course, the greatest benefit is a 7:1 chip count reduction, reducing size and cost while increasing reliability.

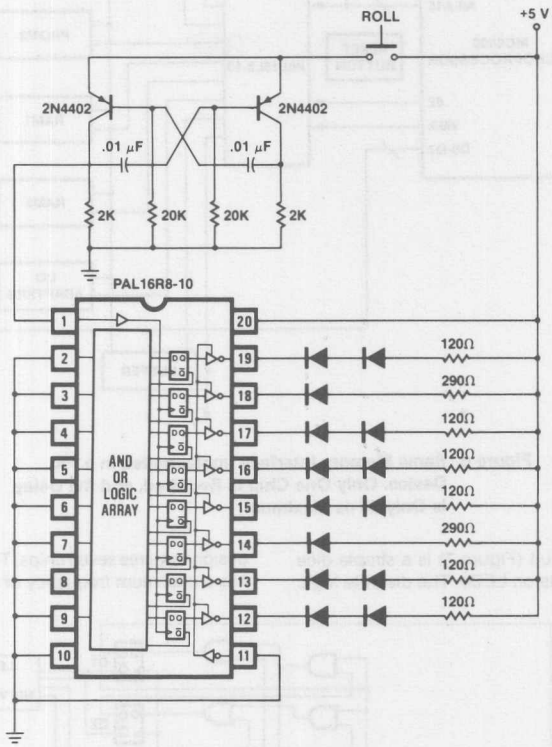


Figure 8. The Same Dice Game Implemented in a PAL16R8-10. Only One Chip Is Required, and the Speed Is Doubled, with Little Increase in Power Requirements.

The PAL 16R8-10 Series devices can also be compared to FAST MSI functions on a one-to-one basis. For example, the clock-to-output time of a standard 74F374 octal register is 10 ns, vs. only 8 ns for the PAL16R8-10.

A straightforward combinatorial example is a simple 3:8 decoder with three-state outputs, the 74F538. This device has a maximum propagation delay of 17 ns, while the PAL16L8-10 can implement the same function at only 10 ns, 41% faster.

Of course, PAL devices add the additional capability of doing custom variations of these functions, if needed. For example, with the substitution of a registered PAL16R8-10 in the 74F538 example, control storage functions such as load and clear can be added.

Even faster discrete logic families have been announced that improve upon the speeds of FAST logic. Many of these families even use CMOS technology. But the fastest devices announced are at 100 MHz, equivalent to the combinatorial speed of the PAL16R8-10 Series. And the PAL16R8-10 Series is available now, providing thousands of functions through its programmability. We saw in the memory interface example that the high-performance logic families are not as complete as earlier families. PAL devices can replace this logic, or complement it with additional functions.

We have shown that the PAL16R8-10 Series creates a new application area for programmable logic: the design of higher performance solutions than any discrete logic implementations. Speed-critical discrete logic designs are now open to replacement by programmable logic.

Elimination of PC Board Noise

This capability may be arriving just in the nick of time. Discrete logic is today achieving speeds that challenge printed circuit board layout. The fast switching speeds can create noise and crosstalk that can bring the high-performance board crashing down. Note the new high-speed CMOS logic families that require new pinouts and extra grounds because of switching noise.

Integrating this logic into programmable logic eliminates not just discrete chips but all those noisy and troublesome connections between them. Putting the PC traces into silicon eliminates crosstalk and signal distortions caused by transmission line reflections, and improves the reliability of the board. In high-speed systems, the length of the interconnect can approach the signal wavelength, causing ringing in the connection line. This ringing can cause a false signal value to appear on an input tied to the ringing line, or can even be fed back into the source device if feedback is used from the output.

Improved Speed

The reduction in interconnect afforded by PAL devices, with the accompanying reduction in inductance and time delays, even increases system speed. In high-speed logic systems, each 12-inch length of interconnect wiring introduces a time delay of approximately 2 ns, about an entire gate delay. These delays slow down the system and can cause glitches and signal errors. Thus, integration of high-speed logic into PAL devices improves not just reliability, but also speed.

Technology Advantages

Two other advantages of the PAL16R8-10 Series, lower cost and reduced signal noise, are a result of the technology used to achieve its high speed. Speed improvements in semiconductor devices come about primarily through size reductions and reduced capacitance. Capacitance, since it stores charge, slows down devices as they have to charge and discharge capacitive elements.

The 25-ns devices were achieved through the reduction of lateral geometries in a junction-isolated process. This produced lower collector-base capacitance and therefore provided higher speed. The 15-ns devices used the same junction-isolated process, but a reduced epitaxial layer produced the higher speeds.

Oxide-Isolated Process

For the 10-ns devices, Monolithic Memories used a completely new oxide-isolated process to increase the speed to record levels. Oxide-isolation allows the transistors to be placed closer together (Figure 9). This greatly reduces lateral geometries while reducing the parasitic capacitance between circuit elements, providing much higher speed.

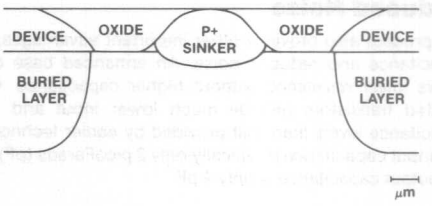


Figure 9. Basic Transistor Isolation Structure in Oxide-Isolated Process. A Fully Recessed Oxide Region Separates Transistors, Allowing Smaller Geometries and Higher Speed.

The process utilizes a walled-base technology. It features 1.5-micron design rules and dual-layer metallization. The isolation technology uses a combination of high-pressure oxidation to form a fully-recessed oxide through the epitaxial layer, and a diffused guard band. This technology avoids any contact between the channel stop and the buried layer.

Unique Needs of Programmable Logic

Programmable logic devices have unique process needs because of the high voltages incurred during programming. The oxide-isolation technique is especially useful because of its excellent insensitivity to substrate injection, which occurs when small transistors are driven into hard saturation during programming.

Lower Manufacturing Costs

Another requirement for programmable products is a good high-current beta so that emitter size can be minimized. The NPN transistors in this process have been measured to have a beta of higher than 20 at an emitter current of 30 mA. In addition, a reduction in the required fusing current allowed smaller transistors in the programming array. Separation of the current sources for programming and normal operation improves the speed-power product. These features add reliability to the product while reducing die size.

High Yield

The walled diffusions improve yield by reducing manufacturing tolerance requirements. In addition, the process is insensitive to threshold problems. The result of these techniques is a high-yielding, reliable process that produces both the 10-ns TTL PAL devices and the 6-ns ECL PAL devices.

Smaller die size and high yields result in a much lower cost of manufacturing. Thus, once the process is mature, it can produce faster products at the same or less cost than earlier technologies. This can be seen with the 15-ns devices, which now are cost-competitive with any other technology, including CMOS.

Many users of ICs claim that it is impossible to find products that combine three elusive qualities: good, fast, and cheap. The advanced technology used in the high-speed PAL products allows them to fit all three requirements well.

Reduced Noise

The process also provides other important advantages, lower capacitance and reduced noise. An enhanced base doping allows lower resistance without higher capacitance. Oxide-isolated transistors provide much lower input and output capacitance levels than that provided by earlier technologies. The input capacitance is typically only 2 picoFarads (pF), while the output capacitance is only 4 pF.

Elimination of False Logic Switching

Lower capacitance allows your system to operate with cleaner switching signals between devices, providing a much more reliable system. This is critical in high-speed systems where fast rise and fall times can cause signal fluctuations.

Guard rings limit the susceptibility of the inputs and outputs to voltage undershoots and overshoots. Bipolar Schottky circuitry is designed with nonlinear input and output impedances. In high-speed systems, undershoots in particular can play havoc with false logic states, if the guard rings do not prevent them.

New Synchronization Applications

Another key advantage of the PAL16R8-10 Series, only indirectly related to speed, is its excellent resistance to metastability. Metastability is a failure that occurs when required minimum setup times are violated. The PAL16R8-10 Series is less susceptible to metastability, and recovers faster, than any other PLD and most other forms of logic. This subject is discussed in detail in Session 3 of Electro/87 on Metastability.

Metastability characteristics are explicitly specified in the PAL16R8-10 Series datasheet (Figure 10). For example, at 21-MHz clock frequency and 7-MHz data input frequency, a metastability failure will occur only once in ten years.

Metastability Characteristics Over Operating Conditions

SYMBOL	PARAMETER	TEST CONDITIONS	COMMERCIAL MIN	TYP	MAX	UNIT
p	Poisson process rate		0.85	1.05		ns^{-1}
k	MTBF constant			0.8	1.0	μs^{-1}
t_{MET}	Minimum recovery time in asynchronous mode	MTBF = 10 years $f_d = (1/3)f$ d = 3		20	30	ns
f_{MET}	Maximum frequency in asynchronous mode	MTBF = 10 years $f_d = (1/3)f$ d = 3	21	26		MHz

Figure 10. Guaranteed Metastability Characteristics for the PAL16R8-10 Series. The High Resistance to Metastability Allows New Use in Synchronization Applications.

Thus, a PAL16R8-10 Series device can be used as a synchronizer circuit, a new application that doesn't even require its programmability. Almost every digital system must at some point handle incoming signals that are not synchronized to the system clock rate, and any such system will encounter metastability. Using the PAL16R8-10 Series devices at the interface provides the lowest possible failure rate.

Lower Metastability Means Higher Speed

For applications using the PAL16R8-10 Series as a synchronizer, the lower metastability rate means higher clock speed. For a given desired MTBF, the maximum clock rate is limited by the susceptibility to metastability failure. The reduced metastability rate for the PAL16R8-10 Series allows a higher clock speed, with no reduction in MTBF.

ECL Provides Highest Speed

The two ECL PAL devices, the PAL10H20P8 and PAL10H20G8, provide the fastest PLDs. 10KH ECL technology also eliminates many of the noise problems encountered with high-speed TTL boards since the internal switching voltage range is less than one volt, and it is often a good alternative to high-speed TTL. ECL design techniques achieve high speed by not allowing the bipolar transistors to saturate when they are on. Thus, there is no storage time delay caused by switching in and out of saturation.

The elimination of saturated transistors means that current is always flowing in ECL devices. This is the cause of high power consumption in ECL circuits. However, the ECL PAL devices consume only 210 and 225 mA for the PAL10H20P8 and PAL10H20G8, respectively. This is only minimally higher than the standard power consumption of TTL programmable logic circuits.

New PLD Applications in ECL Systems

However, these devices do not just speed up programmable logic. These devices are fully 10KH compatible and meant to be used in 10KH systems. They open up new applications for programmable logic in the highest performance ECL systems.

ECL MSI/LSI Functions

Since the ECL market is smaller than the TTL market, fewer functions are available in ECL than TTL. This is true on all levels, from SSI and MSI through VLSI. ECL PAL devices provide a flexible solution that can implement functions not found in fixed ECL logic. Future ECL PAL devices will provide higher density and be capable of implementing the most complex LSI functions.

Systems are built with ECL technology primarily for high performance, and ECL PAL devices must also be high performance. Future ECL PAL devices will improve speeds down to the 3 to 4-ns range and be able to compete effectively with the speed of fixed ECL logic.

The Speed Challenge of CMOS Technology

CMOS technology is quickly encroaching on bipolar speeds. In fact, Monolithic Memories has CMOS mask-programmable devices (ZHAL™ devices) that are even faster than their bipolar equivalents. However, CMOS has not yet reached the best bipolar speeds, and is not likely to in the near future. Even if we exclude ECL from the comparisons, bipolar has a two-generation speed advantage over CMOS technology.

The fastest CMOS PLDs available today are at 25 ns, while 25-ns bipolar devices have been available for years and are currently being shipped in huge volumes. 15-ns CMOS is in development, but bipolar is already at 10 ns and moving further down the scale.

Although recent CMOS developments have been rapid, CMOS is not expected to pass bipolar speeds, only to get closer. As geometries are reduced, CMOS has to contend with short-channel effects and hot-electron trapping as scaling increases the electric field. Bipolar has fewer problems during reduced scaling.

Speed vs. Density

Many vendors of high-density CMOS PLDs claim that they have surpassed bipolar speeds if examined at a per-gate basis. That claim is relevant only if the high densities are required. Even gate array applications are relatively low in density, with over half below the 2000-gate level. Additionally, the largest application area for semicustom arrays is random logic, efficiently met by the PAL architecture. And even in those cases where greater density than a single PAL device is required, two PAL16R8-10 Series devices back-to-back have a combined delay of only 20 ns, still faster than any CMOS PLD of any density.

Some industry observers have raised an issue of speed vs. density, asking which one will win out in the long run. But these two attributes do not compete; they are usually required at different times in different applications, and thus both areas will develop in their own market areas.

Gate Array Performance

Gate arrays and other forms of semicustom logic are demonstrating very high performance, even in CMOS. The sub-nanosecond gate delays of these devices, even when input and output delays are taken into account, cannot be matched by programmable logic. For the ultimate in speed, a more customized design is better, with full custom offering the highest speed.

Programmable logic competes with other high-performance semicustom solutions in applications where the additional advantages of zero NRE, instant customization, ease of design changes, inexpensive and easy-to-use software, standardization, low cost, second sourcing, and other issues weigh in favor of programmable logic.

Studies of gate arrays show that clock frequencies for systems using gate arrays are about equal to microprocessor frequencies; around 8-12 MHz, moving to 16 MHz. But further studies show that 76% of users require at least one very-high-speed (twice the clock frequency) combinatorial path through the chip.

Since programmable logic offers only one speed through the chip, the device speed must match the speed of the fastest required path. This is one reason why high-speed PAL devices are required even if system clock rates are low.

CMOS PLDs can be erasable, a feature bipolar cannot match. Although not erased very often in customer applications, erasability adds to the testability of the device during manufacturing. Also, CMOS programmable cells are smaller than bipolar fuses, allowing a smaller die size. These advantages of CMOS will allow it to dominate, in the long term, the PLD market areas that do not require high speed.

Usable Speed

Just as semicustom arrays offer a greater percentage of "usable gates" than PLDs, or gates used vs. gates provided, PLDs can offer more "usable speed" than semicustom arrays. Although typical gate speeds in custom devices can be below 1 ns, that speed is not relevant if the design and simulation tools do not provide the accuracy required to guarantee proper timing.

Since most of a PAL device is fixed logic, guaranteed maximum delays through the device can be specified for any custom design. In addition, minimum delays for the PAL16R8-10 Series are specified in the datasheet for the first time, allowing accurate system timing. Tools such as PLDMaster from Daisy Systems provide PAL device simulation in a board-level environment. PLDMaster can even help partition designs to take full advantage of device speeds.

The PAL device has another advantage over other custom logic that provides the ultimate in timing simulation—a device can be programmed and immediately dropped into a circuit, to check timing. All of these advantages mean that designers can use more of the PAL device's speed than that of other custom logic, giving a PAL device an edge even at equivalent specs.

CMOS Advantages

CMOS has several advantages over bipolar, depending on the technology used. In fact, CMOS technology has opened up more new applications for programmable logic than any other development, because of several unique characteristics. It can be lower in power, can be erasable, and can be smaller. Some of these advantages are important in some cases but not important in others.

Low power is an important advantage, but Monolithic Memories has been producing half- and quarter-power versions of its products in bipolar, simply by increasing the resistor values. Our PAL16R8B-4 Series devices at 35 ns and only 55 mA are lower in power than many competing CMOS devices at similar speeds. The upcoming PAL16R8H-15 Series at 15 ns and 90 mA will achieve even greater efficiencies. As top bipolar speeds improve, the half-power and quarter-power versions will continue at the highest-performance CMOS speed levels.

CMOS power specifications are often only stated at standby, with the device not operating. Operating power for CMOS rises dramatically with increasing input frequency and can surpass even medium-power bipolar devices (Figure 11). For high-speed applications, CMOS has not shown a power advantage over bipolar.

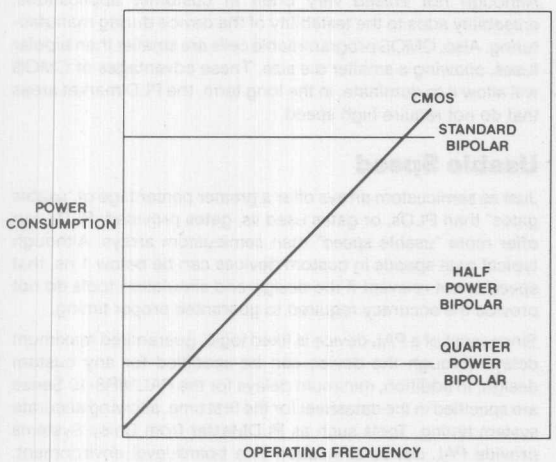


Figure 11. Power Consumption vs. Operating Frequency for CMOS and Low-Power Bipolar PAL Devices. CMOS Devices Can Consume as Much Power as Bipolar Circuits at High Speeds. Thus, for High-Speed Applications, CMOS Loses One of Its Key Advantages Over Bipolar.

Bipolar's Place in Market

Bipolar TTL and ECL PAL devices will continue to dominate the large part of the PLD market that requires high speed. Bipolar will even be dominant in the low-cost part of the market for years to come.

Record Speeds and New Applications

The 10-ns PAL16R8-10 Series and the 6-ns ECL PAL devices provide record speeds for programmable logic, using advanced process technology. Future plans call for 7.5-ns TTL and 3 to 4-ns ECL PAL devices. The speed and technology open new applications for PAL devices, including:

- Highest-performance computers and digital processing
- High-speed microprocessor and memory support
- Speeding up the fastest discrete logic designs
- Component integration, eliminating noise and delays from device interconnect lines
- Process technology advantages of lower cost and reduced noise
- Wide range of LSI functions to complement high-speed TTL and ECL discrete logic families
- Synchronization circuits for low metastability rates

All of these new applications show that these new PAL devices are not just faster devices for the same sockets, but rather open up whole new applications for programmable logic devices.

Bibliography

Sing Wong, Johnny Chen, Kit Gordon, and Scott Graham, Monolithic Memories, Inc., "A 7.8-Nanosecond TTL PAL Device," *Proceedings of the 1986 Bipolar Circuits and Technology Meeting*, September 11-12, 1986.

Steve Zollo, "Fastest Programmable Logic Has 10-ns Propagation Delay," *Electronics*, April 14, 1986.

*Kenneth Chan, Monolithic Memories, Inc., "ECL Technology Suits High-Speed Logic Systems," *EDN*, January 1986.

Ken Marrin, "Advances in CMOS and ECL Process Technology Yield Powerhouse ICs," *Computer Design*, December 1986.

*Kenneth Chan, Monolithic Memories, Inc., "High-Density Interfaces Aren't Superfluous Anymore," *Computer Design*, September 1, 1986.

Roderic Beresford, "A Profile of Current Applications of Gate Arrays and Standard-Cell ICs," *VLSI Systems Design*, September 1985.

*Danesh M. Tavana, Monolithic Memories, Inc., "Metastability," 1984.

Lindsay Kleeman and Antonio Cantoni, "On the Unavoidability of Metastable Behavior in Digital Systems," *IEEE Transactions on Computers*, January 1987.

*Monolithic Memories, *PAL/PLE Device Handbook*, Fifth Edition, 1986

Fairchild, *FAST Data Book*, 1985.

* Available from Monolithic Memories, Inc.

ADVANCED MICRO DEVICES' NORTH AMERICAN SALES OFFICES

ALABAMA	(205) 882-9122
ARIZONA	(602) 242-4400
CALIFORNIA,	
Culver City	(213) 645-1524
Newport Beach	(714) 752-6262
San Diego	(619) 560-7030
San Jose	(408) 249-7766
Santa Clara	(408) 727-3270
Woodland Hills	(818) 992-4155
CANADA, Ontario,	
Kanata	(613) 592-0060
Willowdale	(416) 224-5193
COLORADO	(303) 741-2900
CONNECTICUT	(203) 264-7800
FLORIDA,	
Clearwater	(813) 530-9971
Ft Lauderdale	(305) 776-2001
Melbourne	(305) 729-0496
Orlando	(305) 859-0831
GEORGIA	(404) 449-7920
ILLINOIS,	
Chicago	(312) 773-4422
Naperville	(312) 505-9517
INDIANA	(317) 244-7207
KANSAS	(913) 451-3115
MARYLAND	(301) 796-9310
MASSACHUSETTS	(617) 273-3970
MINNESOTA	(612) 938-0001
MISSOURI	(314) 275-4415
NEW JERSEY	(201) 299-0002
NEW YORK,	
Liverpool	(315) 457-5400
Poughkeepsie	(914) 471-8180
Woodbury	(516) 364-8020
NORTH CAROLINA	(919) 847-8471
OHIO	(614) 891-6455
Columbus	(614) 891-6455
Dayton	(513) 439-0470
OREGON	(503) 245-0080
PENNSYLVANIA,	
Allentown	(215) 398-8006
Willow Grove	(215) 657-3101
TEXAS,	
Austin	(512) 346-7830
Dallas	(214) 934-9099
Houston	(713) 785-9001
WASHINGTON	(206) 455-3600
WISCONSIN	(414) 792-0590

ADVANCED MICRO DEVICES' INTERNATIONAL SALES OFFICES

BELGIUM,		
Bruxelles	TEL	(02) 771 91 42
	FAX	(02) 762 37 12
	TLX	61028
FRANCE,		
Paris	TEL	(01) 49-75-10-10
	FAX	(01) 49-75-10-13
	TLX	263282
Rungis	TEL	(01) 46-87-34-62
	FAX	(01) 45-60-57-25
	TLX	202146
GERMANY,		
Hannover area	TEL	(05143) 50 55
	FAX	(05143) 55 53
	TLX	925287
München	TEL	(089) 41 14-0
	FAX	(089) 406490
	TLX	523883
Stuttgart	TEL	(0711) 62 33 77
	FAX	(0711) 625187
	TLX	721882
HONG KONG,		
Kowloon	TEL	852-3-695377
	FAX	1234276
	TLX	504260AMDAPHX
ITALY, Milano	TEL	(02) 3390541
		(02) 3533241
	FAX	(02) 3498000
	TLX	315286
JAPAN,		
Tokyo	TEL	(03) 345-8241
	FAX	3425196
	TLX	J24064AMDTKOJ
Tokyo	TEL	(03) 207-3131
	FAX	(03) 207-3130
	TLX	232-3390 MMIKKJ
Osaka	TEL	06-243-3250
	FAX	06-243-3253
KOREA, Seoul	TEL	82-2-784-7598
	FAX	82-2-784-8014
LATIN AMERICA,		
Ft. Lauderdale	TEL	(305) 484-8600
	FAX	(305) 485-9736
	TLX	5109554261 AMDFTL
NORWAY,		
Hovik	TEL	(47) 2 537810
	FAX	(47) 2 591959
	TLX	79079
SINGAPORE	TEL	65-2257544
	FAX	2246113
	TLX	RS55650 MMI RS
SWEDEN, Stockholm	TEL	(08) 733 03 50
	FAX	(08) 733 22 85
	TLX	11602
TAIWAN	TLX	886-2-7122066
	FAX	886-2-7122017
UNITED KINGDOM,		
Farnborough	TEL	(0252) 517431
	FAX	(44252) 521041
	TLX	858051
Manchester area	TEL	(0925) 828008
	FAX	(0925) 827693
	TLX	628524
London area	TEL	(04862) 22121
	FAX	(04862) 22179
	TLX	859103

Advanced Micro Devices reserves the right to make changes in its product without notice in order to improve design or performance characteristics. The performance characteristics listed in this document are guaranteed by specific tests, guard banding, design and other practices common to the industry. For specific testing details, contact your local AMD sales representative. The company assumes no responsibility for the use of any circuits described herein.

ADVANCED MICRO DEVICES 901 Thompson Pl., P.O. Box 3453, Sunnyvale, CA 94088, USA
TEL: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450
APPLICATIONS HOTLINE TOLL FREE: (800) 222-9323

© 1988 Advanced Micro Devices, Inc.

B-60M-2/88-0